Lecture Notes on Theory of Computation

Clarissa Littler

August 2, 2014

These notes were originally a series of blog posts I posted on, so they still read that way at the moment. Eventually future iterations will have the lecture notes be better formatted and more like a short book.

1 What is computation?

What is computation? That's a question I'd like to discuss a bit so we can set up the trajectory of this class. Now, wikipedia defines computation as "any type of calculation or use of computing technology in information processing", which is fine but rather dry and not illustrative. A loose, but perhaps evocative, definition of computation would be "anything you can do with a finite amount of data, using a finite number of rules, in a finite length of time". This definition encompasses a variety of thigns such as

- doing arithmetic problems by hand with a pen and paper
- counting on your fingers or using physical objects
- running a computer program

If it's not obvious how all three of those fit my definition of computation, perhaps take a moment to think about how all of them involve finite data, finite rules, and finite time. Here's another interesting example to ponder: is writing a mathematical proof a form of computation?

In this class, we'll be examing various models of computation by trying to answer "how can you tell if a given string belongs to a given language?" Where, "language" means a, possibily infinite, set of strings and by "string" we mean a sequence of characters in an alphabet. Bear with me for a second as we examine what "alphabet" means: an alphabet is just a finite set of symbols that we use as the characters. Why finite? It goes back to our definition of computation: we need to only use finite data in our computations.

Now, if our language L is finite then deciding whether a string w is in L is obviously computable. We can hold the entire language L "in memory" and manually check if w is equal to some string in L. That's finite time, finite rules, and finite data used, so is computable.

What if our language L is infinite, though? Without knowing something *about* L that could give us some hints, it's not obviously something we can compute. We can't have an infinite "lookup table" for a brute force search, and even if we could the process isn't guaranteed to take finite time. On the other hand, if our language is "all finite strings" then this is a simple and computable process: take a finite string as input, then return "true". This is because the language is "simple" to describe, and thus computationally "simple" to test if a string is in the language. On the other hand, let's assume that L is an infinite set of completely randomly chosen strings. There is obviously no simpler description of the language than simply listing the infinite number of strings in the language. Checking that a string is in L is thus not computable.

Most languages, on the other hand, are somewhere between these two extremes. We'll be looking at a sequence of models of computation, starting with DFAs, as we build up to Turing machines, which are a model of computation that makes our "finite time, rules, and data" definition rigorous. We will, by the end of this course, have discussed just exactly what things *are* computable and give you at least some of the tools for showing that something *isn't* computable. We will also tie the "is string w in language L " problem back into a more general notion of computation, revealing that we haven't lost anything by considering such a specific class of problems.

Now, all of that being said there are some necessary mathematical preliminaries that need to be addressed. First off, this will be a very proof heavy course. Sometimes when this course is taught, the assignments tend to be "5 million ways to build a DFA" (apologies to The Coup), but the three of us who are teaching this term are pushing for a more mathematically mature approach.

As such, we should discuss proof techniques. There's three major techniques that are useful in this class: proof by construction, proof by contradiction, and proof by induction. We'll tackle each of these in turn.

Proof by construction is when you show that a theorem is true by constructing some artifact that manifestly demonstrates the property. For example, if we had the theorem "there exists a number greater than 10" a very easy way to prove this is to say "$10 + 1 > 10$, therefore there is a number greater than 10$. An example of proof by construction that we'll see later and repeatedly in the course is showing that some class of languages is closed under properties such as union or intersection by constructing some machine that recognizes the new language.

Proof by contradiction is when you prove that a theorem is true by assuming that it is *not* true, then showing that this contradicts some other fact that we already know is true. For example, let's prove that there's an infinite number of natural numbers. Assume that this is *not* true, then there must be a *largest* number which we'll call n. Since the natural numbers are closed under addition, then we know that $n + 1$ is a natural number that is *bigger* than n contradicting the assertion that n is the maximum number and thus contradicting the assertion that the natural numbers are finite.

Finally, we come to proof by induction. You have likely seen natural induction in some class prior to this. Natural induction states that if we have a property we're trying to prove, $P(n)$, which is indexed by natural numbers, then if we can prove $P(0)$ and $P(n) \to P(n+1)$ then we've prove $P(n)$ for all n. In other words, if we can show that the theorem is true for 0 and that if it's true for one number then it's true for the next number, then we can combine these two properties to show that the property is true for any number. Proof by induction is a far more general principle though and works for many types of data, such as binary trees or lists. In every instance, though, we'll have the same structure for the proof: there will be "base cases" (like $P(0)$) and there will be "inductive cases" (like $P(n) \to P(n+1)$), the "inductive cases" being the ones where you're allowed to assume the property is true for some "smaller" piece of data that you can use to make a "bigger" piece of data. As we come to different examples of inductive data, we will make the rules for an inductive proof over that data very explicit.

2 NFAs, DFAs, and Regular Languages

Today we begin the class in earnest and come back to our initial class of problems: "does the string w belong in the language L"?

We start with a very *simple* class of languages, defined by a very *simple* class of machines called deterministic finite automata (DFA). Pictorially, a DFA is very simple: it's a graph where there is one node that is designated as the *start state*, there are zero or more nodes designated as the *accept states*, and there is exactly one line out from each node per letter of the alphabet.

As an example, consider the following DFA: (insert DFA for (00)*)

How do we *execute* a DFA, though? Being very informal, we say that a string w is accepted by a DFA D when there is a path from the start state to an accept state, whose labled transitions "spell out" w.

As a useful example, trace out how the DFA above computes on the strings "000000" and "000". You should find that you end in an accept state for "000000" but not "000".

Now, in a more formal sense a DFA is a tuple of $(Q, \Gamma, \delta, q_0, F)$ where

- Q is the finite set of states.
- Σ is the alphabet, which you might recall from last time means that it must be finite
- $\delta : Q \times \Sigma \to Q$ is the transition function that defines what the machine does when it receives an input character.
- q_0 is start state of the automata
- $F \subseteq Q$ is the set of accepting states.

In this more formal description, what does it mean for a string to be accepted by a DFA? A string $w = w_0 \ldots w_{n-1}$ \$ of length n is accepted by a DFA when there is a sequence of states $r_0 \ldots r_n$ such that

- $r_0 = q_0$
- \$$\mathrm{r}_{\mathrm{i}} = \delta$ ($\mathrm{w}_{\text{i-1}}$,$\mathrm{r}_{\text{i-1}}$)
- $r_n \in F$

which, in words, says that there's a sequence of states the DFA follows when processing the string and that it ends in an accepting state. Now we can look at the description of deciding whether or not to accept a string and see that it is ultimately a computable process in the sense of the last lecture: there is finite data in the form of the finite states of the DFA, there are finite rules in the form of the transition function δ, and the process of finding what sequence of states the transition function generates on the input takes a finite number of steps when the input is finite. Thus, we can say that a DFA D *decides* the problem "does the string w belong in the language L"? for some language L, where by decides I mean that it always finishes in finite time and gives a "yes" or a "no" answer. A string is accepted or rejected in finite time.

Now, what kinds of languages can be defined with such simple machines? Clearly, any *finite* language can since we can simply create a unique path through the DFA per string in the language, which is possible because there are only a finite number of strings over a finite alphabet so it can only take a finite number of states to construct this automata. However, a notion of computation that can *only* handle finite languages isn't particularly interesting. After all, we know those are computable by lookup table! We'll prove, in the next lecture most likely, that DFAs describe the "regular languages" which, as you might guess, are the languages that regular expressions define.

Let's consider, instead, what the DFAs for a few simple languages look like.

- Σ^*
- $\{"" \}$
- $\{(01)^n | n \geq 0\}$
- $\emptyset$

(insert images later)

Building DFAs for a language is mostly a matter of patience and experience. You learn the patterns for how to do them and get better at seeing whether a DFA correctly accepts the right language. The *act* of building DFAs isn't particularly interesting, so we won't spend that much time on it per se.

Although, as an interesting exercise, let's try building a DFA for the language $\{0^n 1^n | n \geq 0\}$. Can we do it? Does anything seem strange about it? So there's no obvious way to construct a DFA for this language, but does that tell us that there is *no* way to construct such a DFA? No, it doesn't. Instead, in a couple of lectures we'll come to the issue of how one proves a language is *not* regular.

Another thing that I think is interesting to note is that for each regular language, there isn't necessarily only one DFA that can accept it. For example, there are an *infinite* number of DFAs that can describe the empty language, an infinite number of DFAs that describe Σ^*, and so forth with all of the examples we gave above. For the more mathematically inclined, the relationship between "regular languages" and "DFAs" isn't so much an isomorphism as it is an example of an "adjoint equivalence". This is the start of a pattern we'll see for the rest of the course: there isn't a 1-1 relationship between the machines that answer the question "does the string w belong in the language L"? and the class of languages they define.

Now I want to talk about the idea of closure of languages under operations. First we should define what "closure" means. For example, you can add any two integers and get another integer: the integers are closed under addition. On the other hand, if you divide, say, 2 and 3 you do not get an integer: the integers aren't closed under division. A set is closed under an operation when you cannot "escape" the set using the operation. So, we assert that the regular languages are closed under union and intersection. Let us define what these operations are, first:

- $L \cup L' = \{w | w \in L \vee w \in L'\}$
- $L \cap L' = \{w | w \in L \wedge w \in L'\}$

In words, $L \cup L'$ is the language made up of strings in L *or* in L' and $L \cap L'$ is the language made up of strings in both L *and* L'. I've claimed that the regular languages are closed under these operations. How would we show this? Well, we've defined the regular languages as those decided by a DFA. This means that if we want to show that the regular languages are closed under these operations, then we can do so by taking two DFAs M and M' that decide L and L' and then constructing new DFAs $M_\cup$ and $M_\cap$ that decide the union and intersection respectively.

Let's go through somewhat systematically how this construction will work, though we'll elide a proper proof that these constructions are *correct* and instead point you to the book.

Let $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q'_0, F')$ and our goal is to construct $M_\cup = (Q_\cup, \Sigma, \delta_\cup, q_0^\cup, F_\cup)$ and $M_\cap = (Q_\cap, \Sigma, \delta_\cap, q_0^\cap, F_\cap)$. We'll just construct $M_\cup$ at first and then describe how to change it to the $M_\cap$ version.

The basic idea is that we want to simulate running *both* M and M' at once on the input, using our states to keep track of where we are in both DFAs. Then our transition function will operate by stepping us forward in our pairs of states. We can accept whenever *either* M or M' is in an accepting state. This gives us enough pieces we can write out the DFA as a formal tuple. We note, first, that our alphabet is the same this entire time through so we do not repeat it.

- $Q_\cup = Q \times Q'$
- $\delta_\cup(q, q')(a) = (\delta(q, a), \delta'(q', a))$
- $q_0^\cup = (q_0, q'_0)$
- $F_\cup = \{(q, q') | q \in F \vee q \in F'\}$

Alright, hopefully it's clear that this really follows through with that "simulation" plan we explained above. What's nice is that the intersection comes from just changing the "or" in the definition of the accepting states to an "and". Again, we skip over the details of showing that a string is in the union of L and L' iff it is accepted by $M_{\cup}$. The basic idea, though, is that if a string is in the union then it must be in at least one of the languages, and then the simulation will end in an accepting state, and visa versa.

Of course, this wasn't the cleanest construction. Ideally for the union, we'd like to be able to say something like "try L or L' and if one of them works, accept". We can't do that with DFAs as we've defined them, but next time we'll tinker with our definition of a DFA to get a definition of non-deterministic finite automata (NFA) that still decides the regular languages. We'll do some more closure properties, prove that NFAs and DFAs decide the same set of languages, and perhaps work with regular expressions.

3 NFAs and Proving Equivalence of DFAs and NFAs

So, yeah, we'll have to have more examples when I actually do the NFA and DFA lectures. For now, though, I'm going to put up what I have on this post, which ended up being the longest yet, and just keep posting two of these a week and hoping that I'm staying ahead of my real lectures. Those real lectures start next week, by the way, which is a little scary.

Given where we ended last time, we wanted something like a DFA but where an informal description such as "try M or N and if one of them works, accept" might make sense as an implementation of the union for regular languages. Now, what we really want here is the ability to make a *non-deterministic* choice of which branch we take: M or N. We can think of a non-deterministic choice as essentially meaning that we are trying all possible moves simultaneously, and if one of them leads to an accept state then the entire process accepts.

Now, if you look at the picture we want here there's something that might seem a little odd: we don't actually want to consume input as we make this branching move to try running either M or N. This implies that we might want some kind of new move that allows us to move to a state without consuming input. If we combine both of these ideas, non-deterministic choice and transitions that do not consume input we get the definition of a non-deterministic finite automata (NFA).

More formally, we can say that an NFA is a tuple the same as a DFA except that the type of the transition function δ is different. Instead of $\delta : Q \times \Sigma \to Q$ we have $\delta : Q \times \Sigma_{\epsilon} \to P(Q)$ where P is the powerset operator and for any set A then A_{ϵ} is the set $A \cup \{\epsilon\}$ where ϵ is the symbol that corresponds to consuming no input. Now there's a few things we can note here. First, that because the *empty set is an element of the powerset* that we're allowed to have "empty" transitions such as $\delta(q,a) = \varnothing$ which means that in the state q the NFA transitions to *no* states on the character a. This is in sharp contrast to DFAs where there needed to be exactly one transition defined per letter of the alphabet. This allows to, for example, define the NFA for the language that only contains the empty string with only a single state rather than two as follows:

We should also note that we need to change the formal definition of what it means for a string w to be accepted by a NFA N. Recall that previously our definition of computation for a DFA was

"A string $w = w_0 \ldots w_{n-1}$ \$ of length n is accepted by a DFA when there is a sequence of states $r_0 \ldots r_n$ such that

- $r_0 = q_0$
- $r_i = \delta(w_{i-1}, r_{i-1})$
- $r_n \in F$"

Now, looking at the type of our transition function we can see that since δ returns a *set* of states, not a single state, then we need to change the second condition to be $r_i \in \delta(w_{i-1}, r_{i-1})$. This isn't quite right though, as you might have already guessed. We still need to include the ϵ transitions as well! Now, I'll follow Sipser's definition even though I think it's not as clear as it could be. First off, we *define* concatenation of ϵ with other characters as

- $\epsilon w = w$
- $w \epsilon = w$

or in words, that ϵ is the *unit* of concatenation of characters. Then we say instead of $w = w_0 \ldots w_{n-1}$ where n is the length of the string and each w_i is a character in Σ, we instead let $w = y_0 \ldots y_n$ where n is no longer connected to the length of the string and each y_i is an element of Σ_ϵ. Of course, since we've modified our notion of acceptance of a string let's think for a moment and make sure that it's still sensible under our definition of computable. We still have "finite rules" and "finite data", but does it still necessarily take finite time if we're allowing this non-determinism? Consider that one can *simulate* non-determinism with backtracking. We try, sequentially, each possible path for processing the input string. This might end up taking *much* longer based on the possible branching, but since each individual path is finite and the finite number of states means the number of paths is finite, then the sum of all the time needed to try each path is finite. Therefore, NFAs still fit our informal definition of "computable".

All this being well defined, we can perform the union in a very simple way:

which is exactly what we were hoping for in the beginning.

So before we go further into defining regular operations and showing that the regular languages are closed under them, there's a bit of a problem: we have to *show* that the NFAs decide the same set of languages as the DFAs, i.e. that they really are the regular languages.

How would one prove such a thing? Well, what we can do is show that for any DFA M that decides the language L, then there exists an NFA N which also decides L. This would prove that the regular languages are a *subset* of the languages decided by NFAs. The other direction is showing that for an NFA N that decides L, then we can construct a DFA M that also decides L. This would prove that the languages decided by the NFAs are a *subset* of the regular languages. Reminding ourselves that when two sets are subsets of each other, then they are equal, this means that if we can do *both* of these constructions we will have shown that the languages decided by NFAs are exactly the regular languages. This is an example of proof by construction, as we discussed in the very first lecture.

Please note that I'm trying to be careful and say that the set of *languages* decided by DFAs and NFAs are the same. We are not directly comparing NFAs and DFAs or saying that the "set of NFAs" and the "set of DFAs" are equal, because that isn't even a sensible question as they're sets of different "types" of things. On the other hand, they both decide *languages* and we can compare sets of the same thing. In addition, languages are what we really care about here because the set of languages decided tells us about the computational power of a model.

Since we know what construction we want, let's try building it. To start with, the easy direction is showing that for every DFA M that decides a language L there exists an NFA N that also decides L. To do this, first we assume that we have our tuple $(Q, \Sigma, \delta, q_0, F)$ for the DFA M. Now we can make our NFA N as follows

- $Q^N = Q$
- $\Sigma^N = \Sigma$
- $q_0^N = q_0$

- $F^N = F$

and last we have the non-trivial part

- $\delta^N(q, \epsilon) = \emptyset$
- $\delta^N(q, a) = \{\delta(q, a)\}$

or in words δ^N has no ϵ transitions and on a non-epsilon input, it just returns the singleton-set of what δ returns.

This embedding is so simple that as we proceed in the class we may refer to the idea that DFAs "really are" just NFAs. To show that this recognizes the same language, we'd need to show that for a string w there exists a sequence of states $r_0 \ldots r_n$ witnessing that M accepts w IFF there exists a sequence of states $y_0 \ldots y_k$ that witnesses that N accepts w. For this construction, the theorem is trivial because the sequence of states is the same in both cases.

As for the other direction, that will be somewhat more complicated. We'll start with recalling two things we've seen before: that for DFAs we simulated the union by using *pairs* of states as our new set of states and that the transition function represents non-determinism as *sets* of states. Combining these two ideas, we get that in order to simulate an NFA with a DFA the states of the DFA should be *sets* of states of the NFA. This is still a finite set of states because the powerset of a finite set is finite, though exponentially larger. The idea here is that we're "paying" for the cost of the simulation in space, not time, since a DFA will always take time linear in the input string. This linearity is why we can't use the perhaps more obvious trick of "backtracking" to simulate the non-determinism: it doesn't fit the computational model of a DFA.

We can then take a stab at defining the DFA M, given that our NFA is described by the tuple $(Q, \Sigma, \delta, q_0, F)$ we can define our new DFA M as

- $Q^M = P(Q)$, the states of M are sets of states
- $q_0^M = \{q_0\}$, the start state is the singleton set of the original start state
- $\delta^M(qs, a) = \bigcup_{q \in qs} \delta(q, a)$, the transition function takes a step from all its possible states and collates the results into the new set of possible states
- $F^M = \{S | \exists s : S.s \in F\}$, or that our new accepting states are the ones that contain at least one element of the old F. Not *every* state you

can be in needs to be in an accept state, but you need to be in at least one accept state.

Wait, though, there's a bit of a problem here: we haven't taken into account the epsilon transitions. We have to get rid of them somehow in order to have a valid DFA. To do that, we need to introduce a new construct: the epsilon closure of a set of states. The epsilon closure is defined as $E(A) = \{q | q \text{ is reachable from some state in } A \text{ in 0 or more epsilon transitions}\}$, and the reason why it's "0 or more" is that we want $A \subseteq E(A)$ and the "0" guarantees that all elements of A will be in $E(A)$. So given this construct, we need to use it in two places: first, the starting state should really be the *epsilon closure* of q_0 and second in the definition of δ^M we should actually have $\bigcup_{q \in qs} E(\delta(q, a))$. *Now* we have the correct definition of the conversion from an NFA to a DFA.

For this lecture, we'll elide proving that this construction is correct but hopefully it is clear that this follows the prior description of how we'll simulate non-determinism with a DFA.

Here I think I'll end things until my next post.

4 Regular Operations, Regular Expressions, and RegExp/NFA equivalence

Continuing from last time, we've shown that NFAs and DFAs are equivalent. We're now well prepared to discuss what operations *other* than union and intersection that the regular languages are closed under. Having two different ways of representing the regular languages means that we can choose to present our constructions in terms of DFAs *or* NFAs, depending on which is easier.

So other than union and intersection, other operations that the regular languages are closed under are concatenation, Kleene star, and complement. We'll go through and define each of these in turn and prove, *by construction*, that the regular languages are closed under each of them.

First concatenation: we define, for languages L_1 and L_2 the concatenation $L_1 \circ L_2 = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$. In words, the concatenation of two languages is a language that consists of strings of the first language followed by strings of the other. To get some intuition, let's talk about what some simple concatenations are:

- $\emptyset \circ L = L \circ \emptyset = \emptyset$, why? Because there *are* no strings in the $\emptyset$ and thus there is *no* string that can be first or second (respectively) in the concatenation.

- $\{\epsilon\} \circ L = L \circ \{\epsilon\} = L$, because there is only the empty string in $\{\epsilon\}$ and we know that for any $w \in L$ that $w\epsilon = \epsilon w = w$.

  Now, to prove that the regular languages are closed under concatenation we will assume that we have two regular languages L_1 and L_2 and that we have NFAs M which decides L_1 and N which decides L_2. We'll describe in words how we take these two NFAs and make a new NFA that decides the concatenation. First, we take every accept state of M and draw an ϵ transition from it to the start state of N. Then we take the old accept states of M and demote them to regular states. That's it! Pictorially, we can see it as being something like if M is

  and N is something like

then after step one the concatenation looks like

and after step two will look like The next construction we'll look at is complement. Complement is probably what it sounds like, if you have a language L then the complement $\bar{L} = \{w | w \notin L\}$. Now, this might make you uncomfortable a touch. After all, I don't find it inherently obvious that just because you can computably tell if a string is *in* a language that you can computably tell if it's *not* in the language. In the case of regular languages, it's actually pretty easy as we can show using DFAs. If we have a regular language L and a DFA that decides it M, then we can construct a new DFA that decides the complement just by taking the complement of the set of accept states and leaving everything else the same. In other words, if a state was an accept state in M then it's not an accept state in $\bar{M}$ and visa-versa. We can see somewhat intuitively that this is the complement: if a string would end in an accept state of M, then it won't be in an accept state of M' and if it would *not* end in an accept state of M then it *will* end in an accept state of M'. The NFA case is less simple, but the nice thing about knowing that NFAs and DFAs describe *the exact same languages* is that we can use whichever representation is the simplest for our purposes.

Finally, we need to describe the Kleene star. This one is slightly more complicated to describe but very simple to construct. For a language L, the Kleene star is $L^* = \{w_1 \ldots w_n | n \geq 0, w_1 \ldots w_n \in L\}$. In words, the Kleene star operation takes a language and returns a new language that's the concatenation of 0 or more strings in the language. Since "0" is an option, this means that whether or not the language L contains the empty string ϵ, the Kleene star of the language L^* *does* contain ϵ.

We'll show the regular languages are closed under this operation using NFAs. In words, what we do is for our NFA N we attach a *new* state and make it the start state and also an accept state, we make an ϵ transition from the new start state to the old starte state, and then we make ϵ transitions from each of the other accept states to the new start state. Essentially, we are making a loop out of our NFA that can be executed an arbitrary number of times. Why do we make the new start state *also* an accept state though? Well, it's because we've insisted that the Kleene star always include the empty string and this is an easy way to guarantee that our new NFA represents will accept the empty string.

Now given that we have all of these operations, maybe there's another way we can encode the regular languages in a way that is a bit more familiar: the regular expressions. Essentially, the idea of regular expressions is that we describe the entirety of the regular languages with an *inductive* type that includes only things that are obviously regular. So we'll define the regular expressions as being made out of

- a where a is a character in L
- $R \circ R'$ where R and R' are regular expressions
- $R \cup R'$ where R and R' are regular expressions
- R^* where R is a regular expression
- $\emptyset$
- ϵ

Now, intuitively we want the regular expressions to be *exactly* the regular languages. First, though, we should have a way to describe what it means for a regular expression to match a string. We can describe it in terms of *expansions* and we'll do so inductively:

- a expands into the literal character a
- $R \circ R'$ expands into ww' where w is an expansion of R and w' is an expansion of R'
- $R \cup R'$ expands into an *either* an expansion of R or an expansion of R'
- R^* expands into ϵ or it expands into ww' where w is an expansion of R and w' is an expansion of R^*

- $\emptyset$ expands into nothing
- ϵ expands into the empty string ϵ

Now, we say that a string w is accepted by a regular expression R when there exists *some* expansion of R that is equal to the string w. For example, if we have a regular expression $0^* \cup 1^*$ and we want to match it against the string 000 we can expand the regular expressions as follows $0^* \cup 1^* \rightarrow 0^* \rightarrow 00^* \rightarrow 000^* \rightarrow 0000^* \rightarrow 00000^* \rightarrow 0000\epsilon = 0000$. Let's make sure that this notion of "expansion" is computable according to the informal criterion we've been having to use so far. As we can see, expansion only has a finite set of rules so we're good on that front, and since we can terminate our expansion whenever we're out of options or we've exceeded the length of the target string we only need finite data and finite time. This means that our ability to test whether a string is generated by a regular expression is computable.

So while we can intuitively believe that our definition of regular expressions does, in fact, describe regular languages we want to actually *prove* it. In order to prove it, we need do what we did for the DFA/NFA correspondence: we first show that we can take any regular expression and turn into into an NFA, then go back the other direction and take any NFA and show we can convert it into a regular expression that decides the same language.

We'll start, again, with the easy direction: converting a regular expression to an NFA. We'll define this inductively, that is case by case, over the structure of regular expressions.

- a becomes the NFA that accepts the single character a
- $R \circ R'$ becomes the concatenation of the NFAs for R and R'
- $R \cup R'$ becomes the union of the NFAs for R and R'
- R^* becomes the Kleene star of the NFA for R
- $\emptyset$ becomes the NFA for the empty set
- ϵ becomes the NFA for the language that only has the empty string

So, now for the hard direction which is converting NFAs to RegExps. The way we'll do this is with the path $NFA \rightarrow DFA \rightarrow GNFA \rightarrow RegExp$. Gosh, GNFAs aren't something we've seen yet are they? Let's defined them. Informally, they are NFAs where we are allowed to have regular expressions as labels rather than simply characters. The idea being that the transition occurs when some prefix of the input string can be "consumed" as an expansion of the regular expression that labels the transition. We follow Sipser in our insistence that all our GNFAs meet the following conditions

- The start state has transition arrows going to every other state but no incoming arrows
- There is only a single accept state, distinct from the start state, and there is a transition from every other state to it
- Every other state has one transition to every other non-start/non-accept state including itself

  Wow, those conditions might feel kinda weird, but they're meant to make the construction as easy as possible. So the way our construction works is that we can take NFAs to DFAs with the powerset construction we've seen earlier, then we can turn DFAs into GNFAs, and ultimately turn GNFAs into RegExps in a principled way.

5 Finishing NFAs to Regexps, Pumping Lemma and Proving Languages Non-Regular

Continuing from where we left off last time, with the definition of GNFAs, we needed to show that we can take a GNFA with our peculiar restrictions and turn it into a RegExp. Again, we follow Sipser extremely closely. In part, because all of this is tedious enough I didn't feel like trying to be original in my presentation. We start off by taking our DFA M and turning it into a GNFA N as follows:

- Add a new start state with an ϵ transition from it to the old start state
- Add a new accept state with an ϵ transition *to* it *from* each of the old accept states
- Where there are multiple transitions between states of the DFA, we combine them using $\cup$ into a regular expression that matches the "or" of the individual transitions.
- Whenever there are no transitions where the requirements of our GNFA force there to be one, add a transition for $\emptyset$

Alright, from here hopefully it's obvious that M and N recognize the same language given all this graph-surgery. From here, though, we need to progressively construct a GNFA that keeps recognizing the same language until we get one that can obviously be interpreted as a RegExp. What does that mean, you might be wondering? Well the basic plan is that we'll keep simplifying the structure of the GNFA until there are only two states: the start and the accept state, and there will be one transition between them which is labeled with *the* regular expression that matches the language decided by our original M.

We describe the iterative process as follows:

- if there are only two states, then we return the RegExp that labels the solitary transition in the graph

- if there are more than two states, we arbitrarily choose one of them that isn't the accept or start state and "rip" it out. We'll call this state, again following Sipser, q_{rip}. Now, we "repair" the GNFA by, for all states q_i and q_j which are not the accept or start states respectively, we make the new transition from q_i to q_j be $(R_1 R_2^* R_3) \cup R_4$ where $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$, and R_4 is the original transition between q_i and q_j. So what does this mean in words? It means that we are taking into account that there are two ways, now, that we can use to get from q_i to q_j: the original path or the path that went through q_{rip}.

  Since our process removes a state every time, we know that this recursion is well-founded and that we'll eventually terminate. Each step in the algorithm keeps the same meaning in terms of how the regular expression can expand, so the final regular expression returned will correspond to the original NFA.

  It's a bit of a goofy construction, I know, but there's something to be said for going through it in detail so that we have reason to believe that *the* regular languages match up exactly with *the* regular expressions.

Now that we have all these different examples of how to define the regular languages, let's talk about what languages *aren't* regular. Awhile back, we asked if we could define a DFA for the language $\{0^n 1^n | n \geq 0\}$. Of course, we couldn't actually do this but the absence of evidence isn't evidence of absence. We wanted to *prove* that we couldn't ever build a DFA or NFA for this language.

In order to do that, however, we need a tool called the pumping lemma for regular languages. The pumping lemma states that

- For any regular language L, there exists a constant p that we'll call the pumping constant.
- For all strings w such that $|w| \geq p$, then *there exists* strings x,y, and z such that $w = xyz$ *and* $|xy| < p$ *and* $|y| \geq 0$ *and* such that for all numbers $i \geq 0$ then xy^iz is in L.

  Now what does the pumping lemma actually mean? It tells us that for every regular language there must exist *some* size p such that all strings of size p or larger must have some kind of "loop" that can be repeated an arbitrary many times. We can use this to prove that a language isn't regular, by showing that the pumping lemma does *not* hold. If the pumping lemma doesn't hold for a language, and yet the pumping lemma holds for all regular languages, then the language cannot be regular.

  We need to *prove* this lemma in order to actually use it that way, though. We start by noting that since we want to prove this lemma about regular languages, that means we're proving it about languages that can be represented as DFAs. So now we assume that L is a regular language. L thus has some DFA M that decides it. M, being a DFA, has a finite number of states n. We will now prove the pumping lemma with n as the pumping length.

  This argument, essentially, proceeds based off of the "pigeonhole principle". Assume we have a string w, accepted by M, of length l greater than n. Then we know that, since this is a DFA, there must exist a length l sequence of states $q_1 \ldots q_l$ that the DFA passes through. Now, since there are more states in this sequence than there are states in the DFA. This means that, by the pigeonhole principle, that some of these states must be repeated. Since the sequence of states follows transitions, this means that there must be *some cycle* in the graph. If there's a cyle in the graph, then we should be able to repeat that cycle as many times as we want. This cycle corresponds to y in the pumping lemma and the chunk of the string before the start of the cycle is x and the piece of the string after the cycle is done is z. Now, let's check and make sure that we actually are satisfying the pumping lemma:

  For every string with a length greater than n, we know that a cycle occurs in the first n characters because in n characters we must pass through $n + 1$ states, which means that we hit our cycle. As describe above, the part before the cycle, if it exists, will be our x and then the cycle will be y. Everything after the cycle will be z. We have that

$|xy| \leq p$, that $|y| > 0$, and thus we can repeat the cycle so that for all $i \geq 0, xy^i z \in L$.

Neat!

Now we come back to how we should *use* the pumping lemma. Let's consider the following example that we've done in class before: $\{0^n 1^n | n \geq 0\}$. So the pumping lemma says that *for all* strings, then *there exists* a way to break them up into xyz, such that *for all* i $xy^i z \in L$. Now, in order to prove a language *isn't* regular, we start by assuming the language *is* regular and then show that it fails to obey the pumping lemma as follows

- we assume that the pumping length is p
- *we* pick a string s such that $|s| > p$
- in order to show that there exists *no* way to break the string into xyz such that $xy^i z$ is always in the language then we have to consider *all* possible ways s can be broken into xyz such that $|xy| \leq p$ and $|y| > 0$ and then show that no matter how the string is broken up we can pick an i such that $xy^i z$ is *not* in L

for this particular example let's pick

- $s = 0^p 1^p$
- then the way we break up this string *must* be $x = 0^l$, $y = 0^m$, $z = 0^n 1^p$ such that $m > 0$ and $l + m + n = p$. No matter what exactly l, m, n are then we have that $xy^0 z = 0^{l+n} 1^p$ which is *not* in the language

  We'll leave this here for now and continue next time with expanding the languages we can cover to a larger set: the context free languages

6 Context Free Languages, CFGs, PDAs

Now we come to our next notion of computation beyond the regular languages and their associated models of computation, regexps nfas and dfas: the context free languages. Our motivating example is going to be the language we've seen repeatedly at this point, $\{0^n 1^n | n \geq 0\}$. We showed last time there was *no* way to make a DFA that decides this language.

Again, we'll define our set of languages in terms of some model of computation. To this point, we introduce context free grammars (CFGs). A context free grammar is like a regular expression but much more powerful.

The basic model of computation is the same: we have a set of symbols and rules to expand them. What's different about CFGs over RegExps is that RegExps have a pre-defined set of rules for their expansion, meanwhile part of the definition of a CFG is the set of rules for expansion of symbols.

To whit, the CFG that matches our troublesome language is

- $A \to 0A1$
- $A \to \epsilon$

So, for example, we can expand to get the string "00001111" by the sequence of expansions $A \to 0A1 \to 00A11 \to 000A111 \to 0000A1111 \to 00001111$. Let's define these CFGs a bit more formally.

A context free grammar is:

- A finite set of variables V
- An alphabet Σ, where Σ and V are disjoint. These are the "terminals" of the CFG.
- A finite set of rules R, where a "rule" is a pair of a variable and a sequence of terminals and variables.
- A distinguished variable that's the start variable

The set of strings that are generated by all the expansions of the grammar is the language described by the grammar. Again, it's a finite computable process because since there are a finite number of rules and any string we are testing against has a finite length we can simply brute force check through all the expansions of the grammar that are the length of the target string.

We can do a number of other things with CFGs. For example, we could have a CFG for palindromes over an alphabet.

There's one special form for CFGs that we should note specifically, which is Chomsky Normal Form. A CFG is in Chomsky Normal Form whenever it has the following properties

- Every expansion of a variable is either to exactly two variables or a single terminal, i.e. is of the form $A \to BC$ or $A \to c$
- No variable except the start variable can expand to ϵ
- No variable can expand to the start variable

This means that Chomsky Normal Form CFGs have a very simple *inductive* structure that we can take advantage of for proofs. What's particularly useful is that, as we'll show, *all* CFGs have an equivalent CFG in Chomsky Normal Form that generates the same language.

Now we make this construction clear in steps:

- We first introduce a new fresh start variable, S', and have it expand to the old start variable S
- The second step is that remove all rules of the form $A \to \epsilon$. This is a recursive process where we pick a variable A that has an ϵ expansion and then we remove that rule and modify the rest of the expansions to account for the fact that A can expand to nothing. We do this by taking every rule that contains an A on the right hand side, i.e. something like $X \to B \dots A \dots C$, and replace it with a rule that has the A removed, i.e. $X \to B \dots C$. Now, if the rule is $X \to A$ then we replace it with $X \to \epsilon$. Wait, aren't we removing the ϵ transitions? Yes, and so we iterate this process until all rules that have an ϵ on the rhs *other* than the start variable are eliminated. We are, essentially, propagating up the use of ϵ to the top of the derivation tree.
- Next, we replace all rules of the form $A \to B$ by inlining the possible expansions of B so that if we had $A \to B$ and $B \to \dots$ then we replace $A \to B$ with $A \to \dots$. Note that in this step we don't remove expansions from B
- Now, finally, we take care of rules where a variable expands to more than two variables, more than one terminal, or a mixture of variables and terminals. If we have an expansion such as $A \to 0B$ we replace the 0 with a new variable and a single expansion, i.e. $A \to 0B$ will become $A \to XB$ and $X \to 0$. If we have an expansion that has more than two variables, such as $A \to BCD$ then we add a new variable that expands into the sequence piecewise, i.e. the rule becomes $A \to XD$ where $X \to BC$. Note that there's some freedom here but that no matter how you choose the steps involved you'll get an equivalent grammar in Chomsky Normal Form

It's probably a good time for an example, so let's consider our language above for

- $A \to 0A1$
- $A \to \epsilon$

Following step 1 of the above process, we get a new start symbol that must expand to A so the grammar becomes

- $S \to A$
- $A \to 0A1$
- $A \to \epsilon$

now, we eliminate the ϵ transitions.

- $S \to A$
- $S \to \epsilon$
- $A \to 01$
- $A \to 0A1$

You can see that everywhere there was an A on the rhs, we've added a new rule that has the A removed. Now the only place ϵ shows up is in an expansion of the start variable, which is allowed in Chomsky Normal Form.

Next, we eliminate unary transitions so now we have

- $S \to 01$
- $S \to 0A1$
- $S \to \epsilon$
- $A \to 01$
- $A \to 0A1$

Yes, this step has created a lot of redundancy in the rules. Chomsky Normal Form is useful for its simple inductive structure, but the price of simplicity is that we can no longer represent things as compactly as we'd like.

Finally, we put all the remaining rules in the proper form. First, we'll clean up the terminals and then make the rest of rules only expand to two variables.

- $S \to XY$
- $S \to XAY$
- $S \to \epsilon$

- $A \rightarrow XY$
- $A \rightarrow XAY$
- $X \rightarrow 0$
- $Y \rightarrow 1$

and after the final bit of cleanup

- $S \rightarrow XY$
- $S \rightarrow ZY$
- $S \rightarrow \epsilon$
- $A \rightarrow XY$
- $A \rightarrow ZY$
- $X \rightarrow 0$
- $Y \rightarrow 1$
- $Z \rightarrow XA$

and our grammar is now in Chomsky Normal Form. Wow, umm, that's a lot uglier and harder to read now isn't it? Moving on!

So when dealing with the regular languages, we had regular expressions which had an interpretation as DFAs/NFAs. Now if CFGs play the role of regexps for the context-free languages, then what plays the role of the NFA? Let's think for a moment about why we couldn't build an NFA for that pesky language $\{0^n 1^n | n \geq 0\}$. We didn't have any notion of "memory" for our NFA, there was no way to keep count of how many 0s we'd already seen so we'd know to only accept an equal number of 1s.

That being said, if we had something that was *an awful lot like* an NFA yet had a notion of memory then maybe that would solve the problem. That's exactly what we're going to introduce: Pushdown automata (PDAs). We'll get to those next time.

7 Introduction to PDAs

Continuing from last time, we have that the "machine" that corresponds to CFGs are PDAs. Informally, our machines will be finite automata with a limited notion of memory: a stack. In our transitions, we'll be allowed to not only look at the input character when making our decision but we'll also be allowed to look at the top of the stack. When we make a transition, we pop a symbol from the stack and then look at both the next character of the input stream as well as the character we just popped. Note that there's no reason why the input stream and the stack have to have the same alphabet, so in our definition of push down automata we'll allow them to be different. After we make our transition, we will optionally push another character to the top of the stack. A PDA accepts a string when we reach an accept state at the end of processing the string. This is informally computable by the definition we've been using since, because you can only look at the top character our number of rules is just going to be, roughly, the product of the number of states, the size of the input alphabet, and the size of stack alphabet. We can clearly do this in finite time for the same reasons that our NFA and DFA were finite, and we only need a finite amount of data for storing the stack and the state machine. So, this is also a nice computable definition.

One thing we should address: should our PDA be deterministic or non-deterministic? If we think about our goal, which is to have a kind of machine that represents context free languages and has the same power as context free grammars, are context free grammars *inherently* deterministic or non-deterministic? Let's consider a grammar such as

- $A \to 0A$
- $A \to A0$
- $A \to \epsilon$

and let's consider the string 00. How many ways are there to expand the start variable, A, into this string? Just at first blush, I believe there are four different ways. If there's ambiguity in how we generate strings, *how* do we pick? Non-deterministically! Context free grammars are naturally non-deterministic. Now, you might wonder if for every CFG there exists a *deterministic* CFG that also describes the same language and thus the non-determinism isn't necessary. It turns out that, indeed, the CFGs and determinstic CFGs are *not* equivalent. I don't actually know a cute way to demonstrate this, but if I end up finding one I'll share it with the class. (Also, that's a hint to anyone reading this that if they know a cute demonstration that I'm overlooking then please share!)

We'll include the formal definition as a tuple just like we did with NFAs/DFAs. It consists of

- A finite set of states Q
- Σ, the input alphabet
- Γ, the *stack* alphabet
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$
- $q_0 \in Q$ which is the start state
- $F \subseteq Q$ which is the set of accept states

Now, let's talk about what all of this actually means. We have a state machine much like what we had with NFAs, it's non-deterministic as we can see if we look at the presence of the power set in the type of δ, and we have two different alphabets now just as we discussed above. Note, though, that the powerset isn't just over the set of states this time but of the product $Q \times \Gamma_\epsilon$. That's because the choices we have aren't just in terms of which state to go to next, but also in terms of what to do with the stack. Continuing, we interpret Γ_ϵ on the left hand side of the arrow in the type of δ to mean that we're popping a character from the stack, if the stack is non-empty, and looking at it in order to make our decision. If the stack is empty, then we get an ϵ instead of an element of Γ. Γ_ϵ means something slightly different on the right-hand side of the arrow, because that's what we're going to be *pushing onto* the stack. In this case, we're either pushing a character from Γ onto the stack or we're optionally pushing *nothing* onto the stack, in which case we're pushing ϵ.

You might wonder, since we've been trying to keep our informal notation of computation intact so far, if there are any limits to the size of the stack. The answer will be "no", because we'll be using only a finite amount of stack after a finite number of steps, since we can either accept or reject a string after a finite number of steps then we know we'll always be using just a finite amount of memory. We could, in a sense, just assume that there's some size limit to the stack that's hidden from us and behind the scenes for every input the PDA gets configured to set the size of the stack to be larger than we could possibly need for an input of that length. That's a completely valid interpretation of things, mechanically, but mathematically let's just assume that there are no hard limits on the size of the stack and just get comfortable with the fact that we only use a finite amount of it if we take a finite number of steps.

We still need to define, formally, what it means for a string to be accepted by a PDA though. First, we define what the state of the stack is at all times by defining what it is after a step of computation.

- If our stack is cw, where $c \in \Gamma$, and $\delta(q,a,c) = (q',c')$ where $c' \in \Gamma_\epsilon$ then our new stack is $c'w$. Note that we're representing the stack as, essentially, being a string here and reusing the machinery of string concatenation to describe this. We could also introduce a list data structure, but Sipser just uses strings to represent stacks, where the leftmost character of the string is the top of the stack, and represent the empty stack as ϵ. I don't entirely agree with reusing strings for this, but can appreciate the economy of abstractions by introducing as little machinery as possible.
- If our stack is ϵ, and $\delta(q,a,c) = (q',c')$ then our new stack is c'.

Thus, as long as we have an initial definition of the state of the stack, we can understand what the sequence of stack states as the computation progresses are.

We say that a PDA M accepts a string w when $w = w\_0 \ldots w_n$ where $w_i \in \Sigma_\epsilon$ and that there exists a sequence of states $r_0 \ldots r_n$ and stack states $s_0 \ldots s_n$ such that

- $r_0 = q_0$ and $s_0 = \epsilon$, i.e. we start in the start state and the stack is initially empty
- $(r_{i+1}, a_{i+i}) \in \delta(q_i, w_i, a_i)$ where $s_i = a_i t$ and $s_{i+1} = a_{i+1} t$.
- $r_m \in F$

Now that we've done all of that we can go ahead and start working out examples of PDAs and show that, indeed, they can handle the kinds of CFLs we're wanting to do. We'll label the transitions with somethig slightly more complicated than before and all our labels will be of the form "(a,b) -> c$ where "a" is going to be the character we read from input, "b" is the character we pop off the stack, and "c" is the character we're pushing onto the stack. And the reason why those are formatted in ugly ascii rather than latex code is that I'm still not sure how to get latex excepted by the tool I'm using to make the inline graphs. In any case, let's consider what the

PDA looks like for the language $\{0^n 1^n | n \geq 0\}$, our old friend. The basic idea is that we're going to use the stack to track how many 0s we see before we start accepting 1s, pushing a 0 onto the stack per 0 we see in the input stream. We then pop off a 0 for each 1 we see, and then we make sure that the whole stack is empty before accepting at the end of input. Wait, shoot, how do we see if the stack is empty? We do that by pushing a special "start symbol" onto the stack during our first transition, and then by having the transition to the accept state only happen by popping the start symbol back off the stack. Also, a last notational thing is that we'll use e for ϵ. Without further ado,

We can see how this graph implements the algorithm we just saw. Now, what about the palindromes? Let's remember that our CFG for the palindromes was

- $A \to \epsilon$
- $A \to 0A0$
- $A \to 1A1$
- $A \to 1$
- $A \to 0$

Well what we want here is to use the memory of the PDA to keep track of all the characters we saw up until we start accepting the "other half" of the string. Of course, how can you tell *when* you've "seen half" of the string? That's where non-determinism comes in incredibly handy, because we can just make that whenever we want. Now, keep in mind, though, that there's those two transitions that we need to get the odd palindromes as well

- $A \to 1$
- $A \to 0$

because they'll mean that when we make the switch from "first half" to "second half" then we'll need to use an ϵ *or* 1 *or* 0. Let's just see what this looks like

We can see that this follows a very similar structure to the language of matched 0s and 1s and that if we trace out something like the execution for 00100 then it should look something like the following, where we represent the computation as a triple of w which will be what's left of the string to process, s which will be the state of the stack, and q which is the state we're in. So we start out in $(00100, \epsilon, M_0)$ and the correctly terminating trace of the execution becomes

- $(00100, \epsilon, M_0)$
- \$(00100,\$,M_1)\$
- (0100, 0,M_1)\$
- (100, 00,M_1)\$
- (00, 00,M_2)\$
- (0, 0,M_2)\$
- \$(ϵ,\$,M_2)\$
- $(\epsilon, \epsilon, M_3)$
- accept

Neat, huh?

We'll leave this lecture here and pick up next post with a sketch of the equivalence of PDAs and CFGs and a bit on the context free pumping lemma

8 Equivalence of PDAs and CFGs, CFL pumping lemma

So we've introduced PDAs and gone through a few simple examples of them. We've also asserted, repeatedly, that the PDAs are equivalent to CFGs in describing the context free languages. Not we need to make good on that assertion. We'll really only cover one side of the equation in detail, since it's the more mechanically interesting side as it tells us how to convert a CFG where matching can be seen as a proof search problem to a straight forward machine where the computational time is going to be proportial to the size of the input.

So we'll show how to convert a CFG into a PDA. Conceptually, we want to "simulate" the CFG's rules as part of the rules of the PDA. What we'll do is let both *variables and terminals* be a part of our stack alphabet Γ, but our input alphabet Σ will simply be the set of terminals. When we have a variable on the top of the stack we'll pop it and push back on, non-deterministically, the right hand side of one of that variable's expansion rules. Whenever we see a terminal on the top of the stack, we consume it. Finally, when the stack is empty we move to the accept state. Gosh, if we need the stack to be empty at the end of an accept state that means we should push on a special symbol before we begin our computation. Let's call it !. We also need to push onto the stack, before doing anything else, the start symbol of the grammar in order to get the whole simulation primed. This means that we'll have three "main" states, and other states in order to handle the pushing of symbols involved.

An example might help things make more sense. Let's consider, again, our language of matched 0s and 1s. We already know how to make this as a PDA, but let's do the conversion and let's see how it matches up with the direct construction. As a reminder, our grammar is

- $A \to 0A1$
- $A \to \epsilon$

We'll allow ourselves a little bit of a cheat at first, and push *multiple* symbols at a time, and then we'll backtrack and show what it looks like if you take the cheat back out again. Consider it notational shorthand for the real graph! You can see how we pushed multiple symbols at once and had a transition for "A" every time we saw it on the stack. Now let's do a run through in the style of the last lecture where we look at input buffer, stack, and state

- $(0011, \epsilon, M_1)$
- $(0011, A!, M_2)$
- $(0011, 0A1!, M_2)$
- $(011, A1!, M_2)$
- $(011, 0A11!, M_2)$
- $(11, A11!, M_2)$
- $(11, 11!, M_2)$
- $(1, 1!, M_2)$
- $(\epsilon, !, M_2)$

- $(\epsilon, \epsilon, M_3)$
- accept

Since what we're doing is a straightfoward simulation of the the CFG on the stack of the PDA, hopefully it's pretty clear that this will decide the same language as the CFG did. For completion, let's include here what the PDA looks like without our cheat for pushing multiple symbols

As hopefully is clear this is just expanding out the push onto the stack into multiple states that do nothing with the input and simply add symbols onto the stack.

Now, I won't really cover the reverse direction of PDA to context-free grammar. It's not super interesting and spiritually reminds me a lot of the conversion of NFAs into RegExps. We first massage the automata into a particular format that's nice and then build up the syntax of the CFG from the transitions of the PDA. You can look it up in Sipser if you particularly care about it. The important point is knowing that *it exists* and thus PDAs and CFGs are equivalent. The PDA to CFG direction, on the other hand, is interesting because it tells us how to implement CFGs easily as a program.

Finally our last topic on context-free languages: the context free version of the pumping lemma. As before, we'll state it then prove it, then do some simple examples with it.

So the pumping lemma for context free languages states that if a language is context free then

- there exists some number p, called the pumping constant such that
- for all strings w in the language such that $|w| \leq p$, then
- there exists u, v, x, y, z such that
- $w = uvxyz$ and
- $|vxy| \leq p$ and
- $|vy| \leq 0$ and
- forall $i \geq 0$, uv^ixy^iz is in the language

Okay, so this looks an awful lot like the pumping lemma for regular languages except that we now break things up into *5* pieces instead and the "looping" parts occur in two places in the string v and y rather than just one. Why is that? Well, in a sense the more flexible kind of recursion we can do with CFGs that allows us to do more than the regular languages explains it pretty neatly. You don't just have simple loops in the CFLs, which would correspond to productions such as

- $A \to BA$
- $A \to \epsilon$

which would give us the simple kind of xy^iz kinda like with the regular languages, however we can also have recursion that does something like

- $A \to BAC$ or
- $A \to AB$ etc.

and a grammar can mix all of these together. That means that the part of the string that's the "loop" can come before, after, or *both* from the base case of the recursion. That's why we have this restriction that $|vxy| \leq p$ but we can "pump" v^i *and* y^i simultaneously.

The basic idea of the proof is similar to the regular language version, where we take the pumping constant to be some size that forces there to be a repetition by the pidgeon hole principle and then we mercilessly exploit that repetition. What number can we exploit? Well, we don't have states like in the DFA case, but we *do* have a limited number of variables. If we can show that there are a number of expansions larger than the number of variables, then we know that there *must* be a repeated variable in there somewhere.

First, let's look at a property of parse trees for context free grammars: the height of a parse tree is the height of the longest path from start node to ending node, or in terms of strings the largest number of expansions from the start symbol to one of the terminals in the resulting string. If we choose our pumping constant to be $b^{|V|} + 1$, where V is the set of variables and b is the largest fanning of any expansion in the grammar, then we know that the height must be greater than $|V|$, and if it's greater than the number of variables then we know that there must be a repeated variable. Let's call that repeated variables R. Then there is some path in terms of recursion from R back to itself, and we can either cut out that subtree entirely, leaving only the base case of the recursion (x above in our breakup of the string) or you can arbitrarily repeat the subtree "under itself" to pump up the repeated part of the string on either side of x, i.e. the v^i and y^i components.

Let's consider an example before we close the book on this topic. Let $L = \{w|w$ is a palindrome and the number of 0s and 1s are equal$\}$.

Assume our pumping length is p, then we pick our string to be $0^p1^{2p}0^p$. Now, since this string is longer than the pumping length we know that there must be some way to break up the string into u, v, x, y, z such that $0^p1^{2p}0^p = uvxyz$, $|vxy| \leq p$, $|vy| \geq 0$, and for all natural numbers i then uv^ixy^z should be in L. Let's consider all the ways we could break up our string into these pieces. This is a little more complicated than the regular case because we have the freedom to pick u to be as long as we want rather than having the loop be forced to occur in the *first* p characters of the string. There are three proper cases

- vxy occurs entirely in the first or last 0^p, but then pumping means that we'll break the invariant that it's a palindrome
- vxy is a mixture of 0s and 1s, but since it can only be wide enough (at most p width) to grab 0s from one side, hence pumping will make it no longer a palindrome
- vxy is made up of entirely 1s, but then pumping can keep the string a palindrome but *can't* make the string still have an equal number of 0s and 1s.

and thus we've shown that the language is *not* context free.

Well, that pretty much wraps it up for everything we're intending to cover about context free languages in this course. There's plenty more to say, really, but it's mostly in the context of parsing or how linguists use them which is all pretty wide outside the scope of this course where we just want to treat them as an intermmediate model of computation. Onward to Turing machines! (which are chronologically before these notes, but wevs)

9 Informal Introduction to Turing Machines

(This lecture is going to appear out of order, unfortunately, but I was incredibly bored of PDAs and I really wanted to still get a post and some writing done to hit my word count goals. I'm serious, y'all, if you think pushdown automata and constructions on them are kinda boring as students just imagine trying to get up your enthusiasm about lecturing on it!)

Finally we come to Turing machines, which are the main construction we've been building up to this entire time. Unlike the machines we've been dealing with previously in this course, these will encompass the *entirety* of the computable functions. In a sense, honestly, we *define* computability based off of what can be done by a Turing machine since it's the most general model of computation we have.

Now this might seem circular to you, since up til this point we've been trying to define exactly what languages can be described by different forms of machine and *now* we just throw in the towel to say that "golly gee whiz, this must be as strong as it gets"? Well not exactly, because as we'll talk about briefly while it's only a *hypothesis* that the entirety of the computable functions are described by the Turing machines, it's a hypothesis that has a lot of evidence going for it. Namely, that every other notion of computation humanity has ever been able to devise is either *equivalent* to Turing machines or, in fact, is a subset of what the Turing machines can do. The lambda calculus, the partial recursive functions, etc. are all equivalent to Turing machines. We know this since we can write implementations of Turing machines in these other computational models **and** we can simulate these other models in Turing machines as well. Just as with our DFA/NFA equivalence or our PDA/CFG equivalent, we know that if we can provide constructions going "both ways" between two different models of computation then we know the models of computation are equivalent in power. The extension of this observed fact to the conjecture that all computational models that attempt to capture the set of all computable functions will be equivalent to Turing machines, and hence to each other, is called the "Church-Turing Thesis".

One might question, though, "what about quantum computers?" and that would be a very good question given what I'm currently asserting. The reality is that quantum computers *can't do anything more than a Turing machine.* We can see this by the fact that we can simulate quantum computers on ordinary classical-mechanics inspired computers we know and love. It would seem, honestly, since we can perform simulations of physics on computers to any observable accuracy that, maybe, all physical processes are in a sense computable. This has actually been hypothesized before, but again there's no real evidence beyond coincidences and gut instincts for any of these things. Who knows? Maybe there will be a discovery some day that will show the Church-Turing Thesis wrong. Your humble lecturer doesn't find this *terribly* likely though.

Given that lengthy preamble, we now come to what Turing Machines *actually are.* As usual, we'll describe it informally first. To start, let's imagine having a machine that's like a DFA-with-scratch-paper. We only have a finite number of states, as usual, but we have a mechanism for looking at the scratch paper, moving across the scratch paper, and writing on the scratch paper. Imagine that the scratch paper is graph paper: made up of cells into which the data is neatly written, which makes it different than every piece of graph paper I've used in my life. When we take a computational step, we're allowed to look at both our internal state and at the particular cell of graph paper that our machine was pointed at at the beginning of the step. When the machine has decided what to do next, it can make any needed notes in that cell of the graph paper and then move the reader to an adjacent cell of the graph paper. Configuration of input to the machine will be done, rather than with some magical input like for DFAs and PDAs, by giving the machine a piece of scratch paper that already has some data on it. For example, if we have a machine that will do arithmetic problems, then the initial state of the scratch paper will be a problem such as "3*6 + 5 = " and then we'll use the scratch paper below the equation to actually figure out *what* "3*6 + 5" reduces to and then at the end of the process write in our answer on the right hand side of the equation.

Now, how much scratch paper do we actually have to work with? This is a slightly delicate question when it comes to making sure we're being "computable" by our informal definition. Let's assume, for the moment, that our machine will always stop in a finite amount of time. Then, since it can only move one step across the scratch paper per step then we have a bounds on the amount of graph paper we need: it will use a number of cells less than or equal to the amount of steps the machine runs. If we know, then, that the machine will halt no matter on what input it's given we know that it will always take a finite amount of paper even if that amount of paper is arbitrarily large. So, we'll assume an "infinite" supply of paper because if all's going well we'll only use a finite amount of it anyway. The supply of paper is infinite in the way natural numbers are infinite, not the way the real numbers are infinite. There may be an infinite *quantity* of natural numbers, but the process of building a single one of them is finite. This discussion might seem nitpicky, but given that we've been insisting on finite data and finite rules this entire time I think it's important to argue that we're not violating those principles that have gotten us this far.

What we've described is, essentially, a Turing machine albeit perhaps one a bit more flexible than we really need. Just to make things even simpler when describing our state machine and what it does, we'll assume a 1-dimensional scratch paper, which by convention is always called tape. This tape will have cells on it just like our graph paper did and, instead of being able to move everywhere on the graph paper, we'll be restricted to just moving left and right on the tape.

Where does this idea come from? Fundamentally, it's inspired by the good-old-days when "computer" was a job title, not an inanimate object, and a job mostly done by women I should add. A computer was someone who did tedious but important calculations for a living, more or less, often employed by the military. Turing's idea was inspired by the fact that when computers were doing their work they always used a finite amount of scratch paper and a computer could take a break and then eventually come back to her work and continue it. The fact that you could take a break and come back to your work without error, in a sense, means that you must be relying somewhat on your own internal memory but also that you're looking at where you were in the calculation. These people were doing calculations that they all understood how to do, and that there was some *set of rules* for how they proceed.

We'll continue next post with the *formal* definition of Turing machines. We'll walk through examples of Turing machines, talk about different levels of descriptions for Turing machines, and maybe even talk a little bit about the rather depressing life of Alan Turing himself.

10 Formal Definition of Turing Machines

Now we get to the formal definition of Turing machines. The formal definition of Turing machines is much like the other machines we've seen so far: there's a state diagram, a notion of transition, and other things that can happen during that transition. Let's describe Turing machines by formal tuples the way we have before, so a Turing machine has:

- a finite set of states Q
- an input alphabet Σ
- a tape alphabet Γ which must actually be a superset of Σ this time because the only way we get input is off the tape. Since scratch paper can be blank, we also insist that $\sqcup$, the "blank" symbol, also be a part of the alphabet Γ.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ which we can read, in words, as saying that δ looks at the current state and the input in the cell of the tape that the reader is currently on, then moves to a new state, writes a new symbol onto to the tape, and then moves the head either left or right. Now, what if we don't actually want to change the symbol on the tape? In that case, we should just write the same letter that was already there back down to the tape. We just, for simplicity of definition, insist that there be only one case for the type of the function rather than multiple

possibilities. Similarly, we insist that we move left *or* right rather than allowing ourselves to stand still just because it simplifies definitions.

- a start state q_0
- an accept state q_a
- a reject state q_r

Note that for the first time we have an explicit reject *and* accept state. Huh, that might seem a bit odd actually. We'll come back to that in a moment. Let's also note that this is a *deterministic* definition. That might seem like a step back from the non-deterministic machines we've been considering the past few weeks, but the reality is that Turing machines have equal power when deterministic or non-deterministic. The only real difference is once we start talking about the efficiency of the machines, and then the distinction matters greatly. Putting all those issues aside, let's figure out what it means for a Turing machine to compute. We'll talk about configurations of Turing machines to do that. A configuration is a combination of a

- the current place of the reader
- the current state
- the state of the tape

Following Sipser a bit, we'll say that a configuration C_1 yields a configuration C_2 if the Turing machine steps from C_1 to C_2 in a single step. Sipser says "can step from C_1 to C_2 in single step", but the word can isn't necessary at the moment since we're dealing with only deterministic machines. There are no choices in this fascist model. We also should define the *start* configuration given a particular state of the tape $w : (0, q_0, w)$. Here we'll be using the fact that a state of the tape can also be read as a string that extends rightward from the start of the tape. We'll also say that an accepting configuration is any triple (n, q_a, w), i.e. one that has the accept state q_a as its current state. We'll similarly call a rejecting configuration any triple (n, q_r, w).

Finally, we can say that a Turing Machine M accepts a string w when there exists a sequence of configurations $C_0 \dots C_n$ such that

- C_0 is the start configuration for the string w
- for every i, C_i yields C_{i+1}
- C_n is an accepting configuration.

and we define a Turing machine M *rejecting* w when there exists a squence of configurations $C_0 \ldots C_n$ such that

- C_0 is the start configuration for the string w
- for every i, C_i yields C_{i+1}
- C_n is a rejecting configuration.

Okay, cool, we have our notion of computation now. Looking at these definitions, we can see that as soon as we hit an accepting configuration or a rejecting configuration then we're **done**. This isn't like a PDA or NFA where we can be in an accepting state and then move out of the accepting state when attempting to process more input.

Perhaps you find this unsatisfactory: maybe you want there to be some notion of being "done" with the input in order to accept, the same way we could think of our previous machines as having an input buffer it consumes. Well this goes back to the inspiration for Turing machines: working out calculations with pen and paper. Think of taking a midterm: you have space in which you're performing the work for the problem, and you don't erase it all before you call it done and hand it in. That's what we're doing here with Turing machines. We *could* require that the tape be blank and the head reset in order to accept, but that would just involve taking ordinary Turing machines and then adding a couple of extra states to handle the cleanup. So let's just skip all of that and say that accepts are accepts, and rejects are rejects, much like what scripture tells us. (My apologies, but once a southern Christian, always with the bad jokes)

Alright, we can't avoid the question any longer, can we? Just *why* exactly do we need both an accept state and a reject state, when we could always have "reject" be the abscence of acceptance before? Let us consider a Turing machine with the following transition function

- $\delta(q_0, a) = (q_0, a, R)$ for all characters in the alphabet a

What does this do, in words? Gosh, it looks like it will just ignore the input and move the head to the right, *forever*. That means it will never *halt*. What about all that business of saying that computation should be done in finite time? Have I been lying this entire time? Let us say that I have been subtly simplifying a question all along.

I've tried to be very careful and say "this machine *decides* this language$ the entire time. We're coming to the distinction between *decides* and *recognizes*. So we'll say that a Turing machine *decides* a language L when, for any string w, the Turing machine will always either accept or reject w, which means that it will tell us "yes" or "no" in finite time. All the machines we've seen so far are of this "deciding" kind: they say *yes* or *no*.

A Turing machine *recognizes* a language L when, for any string w, if w is in the language then the machine will reach an accept state. If w is *not* in the language on the other hand, it might reject or it might run forever, i.e. have an infinite loop. These are also computable operations and, indeed we need to slightly amend our description of "computability" to say that an operation is computable if it, when given well-formed input, will finish in finite time, using finite rules, and finite data.

You might think that that seems kinda awful: we've sullied our nice notion of computation to include non-termination. Well, sadly, the problem is that there are a lot of things that are Turing *recognizable* but *not* Turing decidable. Over the next week we'll see a number of examples of them. Suffice it to say, for now, the idea is that there are many more things you can do computably once you only have to consider well-formed input and just do *whatever* on badly-formed input. That might seem counter-intuitive, but it's strange and true and maybe kinda amazing when you get right down to it. Math is weird!

I don't know if any of you have ever questioned *why* it's even possible to write infinite loops in programs, given that you never actually want to do that. (Note that by infinite loop here I mean one that doesn't "do" anything: an operating system or a server doesn't count as an "infinite loop" for these purposes, but talking about why that is is a touch beyond the scope of this course.) This distinction between recognizable and decideable is exactly the reason: if you want to be able to describe all possible computable functions, then you have to allow for the possibility of infinite loops. It's a tradeoff.

Indeed, there are actually languages such as Coq or Agda that *don't* allow for infinite loops. They can guarantee that every program will actually terminate, which is a wonderful thing to have for many reasons, but there are some programs that they just can't express, even if they're written correctly. (Some more technical people who might be reading this blog might nitpick with that statement, as you can "fake" having all computable functions by using coinduction with a non-termination monad. I'll admit that that's super cute and I love that trick but it's not quite the same thing as the program being a first-class term in the language.)

11 More Examples of Turing Machines and Turing Machine Variants

We've talked enough now about Turing machines in the abstract, now let's talk about how we're going to specify them in this class. To be completely formal, one should should always define the full state machine, but that's not going to be how we actually do things for the most part. We're going to, in general, give an *informal* description of Turing machines by writing out in words what the algorithm does. First, though, let's take a couple examples straight out of Sipser as state machines then we can discuss some intuition for what informal descriptions actually make sense for Turing machines.

First, there's the language $\{0^{2^n} | n \geq 0\}$. Now the idea of the algorithm is that we'll scan across the tape and cross off half the 0s on the tape each time and if we never hit an odd number of 0s before we cross everything off, then we accept, otherwise we reject.

The informal description from Sipser is

"On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained only one 0, accept
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

and the state diagram is

Now, we can see that the state diagram implements the spec of the informal description and our conceptual outline. State M3 is the state where we've seen one zero, and if we see another one then we cross it off and go back into state M2, otherwise we get to the other end we go to the error state. If we keep seeing even numbers of zeros until we hit the edge of the tape, then we go back to state M1: that control flow is what M4 does.

So, the level of description we're going for here is something like the one for Sipser for this problem: a list of steps that are allowed to refer to each other, where you can do things like

- sweep across the tape
- read symbols and change them

- jump to other stages depending on what you read on the tape

If you can describe your algorithm informally without using more complicated concepts than that, then you should be sticking to things that we can implement obviously in a Turing machine. We'll expand these limits a little bit as we establish a set of things that we *know* Turing machines can do, which we can reference like they're predefined functions in a programming language. For example, the ability to simulate another machine given its description as a string and the proposed input, is a computable process. Once we've shown that, we'll have other Turing machines whose informal description will say "Simulate the machine *blah* on the input".

Let's talk about a few other Turing machines that decide languages we've seen before that were not regular or context free. First, we have the arithmetic language $\{m + n = p | m, n, p$ are binary numbers and $m + n = p$ as numbers$\}$. This was not regular or context free by the respective pumping lemmas for those classes of languages. Now, as an informal description of a Turing machine we have something like

On input w,

1. scan to the end of the input and place a # symbol, scan all the back to the left

2. scan repeatedly to ensure that there is a $+$ before an $=$ and that the string before the $+$ and the string before the $=$ are the same length (this can be done by marking them specially to ensure that we've scanned all the appropriate symbols and then in the last step replace the marked versions of the 0s and 1s with the normal versions as a cleanup phase)

3. look for the non-x character closest to the $+$ symbol but to its left, x it out, then scan to the right until non-x character closest to the $+$ symbol but to its left, x it out, and move to the first blank space to the right of the # and, if both symbols were 1s then write a 0 and go to stage 4, if one was a 1, write a 1 and go to stage 3, if both were 0 write a 0 and go to stage 3. If all characters (other than $+$) the left of the $=$ are x'ed out, then go to stage 5

4. look for the non-x character closest to the $+$ symbol but to its left, x it out, then scan to the right until non-x character closest to the $+$ symbol but to its left, x it out, and move to the first blank space to the right of the # and, if both symbols were 1s then write a 1 and go to stage 4, if one was a 1 write a 0 and go to stage 4, if both were 0

then write a 1 and go to stage 3. If all characters (other than +) the left of the = are x'ed out, then go to stage 5.

5. scan back and forth across the # and ensure that the string between the = and # is the mirror of the string to the right of the #, this is just an iterated scan where you mark off with an x matching pairs of characters until everything is x'ed out, then accept. If at any point you are not matching characters then reject.

So that's an informal description of how a turing machine can handle very basic arithmetic problems. We can play similar, but messier, games to describe other operations such as multiplication, division, etc. Note that the key was that we had two stages corresponding to whether or not we had a carry bit in the next step of our add. Hopefully the way this worked made it clear that we, in essence, answered the question "does m+n=p?" by actually *computing* m+n and then checking it against p.

Now, another language we can describe as a Turing machine that wasn't a CFL is the language $L = \{w | w \text{ is a palindrome and the number of 0s and 1s are equal}\}$. This is, basically, just done by scanning to ensure that it's a palindrome and rather than x'ing out the symbols we read we replace them with a "marked" version of 0s and 1s, and if it *is* a palindrome then we scan across to make sure that there are an equal numbers of 0s and 1s by just x'ing out one of each on each pass across the string. It's not the simplest thing in the world, but it works!

Now there's a couple of variants of Turing machines that we should discuss before we move on. First, what if we allowed ourselves multiple tapes to work with? Is that going to be more or *less* powerful than a single tape Turing machine? By power, I mean can it decide and/or recognize the same set of languages, I don't mean *efficiency* which is a separate concern. Well, it turns out that multi-tape machines are just as powerful as single tape Turing machines. Obviously, a single tape Turing machine is a special case of a Turing machine with a fixed number of tapes so we know that the languages described by single-tape machines are a subset of the languages described by multi-tape machines. As for the other direction, we can simulate a multi-tape machine with k tapes by having the contents of all k tapes split up into k regions on the single tape and we move back and forth between them, remembering in states the last character we saw as we move to the next tape segment. If you're familiar with the concept, this is much like "currying" when it comes to functions: a function of two arguments $f : A \times B \to C$ can also be thought of as a function $f' : A \to B \to C$, so similarly we're changing the decision process so that the states of our original TM, rather than taking in all k-arguments at once from k-states, take the k-arguments one at a time, leading to new states each time. This rather dramatically increases the number of states we'll be using in the simulation, and increasing the number of steps as well, but *that's okay* since we just need to know that the simulation is possible.

The other, similar, variant of the Turing machine is the non-deterministic Turing machine. Non-deterministic Turing machines do exactly what they sound like, having multiple possible transitions for each combination of state and symbol on the tape. We can simulate a non-deterministic Turing machine readily enough using a 3-tape Turing machine, which we thus know is equivalent to some single tape Turing machine. The basic idea is that we have a tape for the original input, we have a tape that acts as the working tape for the simulation of a path through the non-deterministic machine, and we have another tape that keeps track of where we are in breadth-first search through possible paths in the computational tree. Now, why breadth-first? Because we want to be certain that if there *is* a path to an accept or reject state, that we find it. Depth-first search runs the risk of diving down a loop when there was a perfectly good terminating path.

I think that'll be all for this post, and next time we'll actually get to talking about what kinds of languages are decideable, what ones are recognizable, etc.

12 Machines Simulating Machines, Some Decideability

Last time we mentioned casually the idea that Turing machines could simulate other Turing machines. This isn't covered much in Sipser, at least not in a way I liked, so let's talk a bit informally about how such a thing makes sense. First off, let's note that a Turing machine itself can be given some textual, finite, description as a string. Thought it might seem silly, remember

```
digraph M {
 rankdir=LR;
 size="8,5"
 node [shape = circle];
 M0 -> Mr [label = "_ -> R"];
 M0 -> Mr [label = "x -> R"];
 M0 -> M1 [label = "0 -> _,R"];
```

```
  M1 -> M1 [label = "x -> R"];
  M1 -> Ma [label = "_ -> R"];
  M1 -> M2 [label = "0 -> x,R"];
  M2 -> M2 [label = "x -> R"];
  M2 -> M4 [label = "_ -> L"];
  M2 -> M3 [label = "0 -> R"];
  M3 -> Mr [label = "_ -> R"];
  M3 -> M3 [label = "x -> R"];
  M4 -> M4 [label = "0 -> L; x -> L"];
  M4 -> M1 [label = "_ -> R"];
}
```

and there's also the graphviz source for drawing that diagram. Now, that source for drawing the graph includes some data for how things should look, but other than that it's a finite text description *of* the Turing machine. Now since we know that a Turing machine can have multiple tapes, let's imagine a machine that has two tapes. The first tape will contain the text description of the machine we're simulating, the second will contain the input to the simulation machine. The basic approach is that we'll step through the simulation by reading the input tape and treating it as normal and then using the tape with the description of the machine to both keep track of what state in the simulation we're in and to figure out what to do at each point. Now, you might object and say that it's not obvious that we can do the appropriate lookups and moving around for an arbitrary number of states in a finitary way. I think it *is* possible, with a clever encoding, to write down the description of the Turing machine so that we go either left or right into the appropriate state on the tape which eliminates the need for a lookup, but means that we need a number of repeated copies of the states on the tape. On the other hand, we could do a naive encoding on the machine description and then just build the machine so that it does a lookup but for only a number of states up to some cutoff. We can just do different versions of the simulator for different "sizes" of Turing machines. In either case, everything will be nice and okay and finite.

There's a strange and important lesson here that I'd like to expound upon for a bit. When we write programs, we're *writing finite text descriptions* of algorithms. Of course, I know you all know that you've been writing text but let's let that sink in for a moment. All our programs are of finite length, the alphabets we use to write the programs are finite, and thus how many programs are there? *Countably* infinite, which in the grand scheme of mathematics is a pretty tiny number. On the other hand, there are *un-*

countably infinite real numbers. That alone pretty much guarantees that we can't easily have exact arithmetic with real numbers. The fact that there are *uncountably* infinite functions $\mathbb{N} \to \mathbb{N}$ also tells us that we're giving up a lot of possible functions by requiring that we can write finite descriptions of functions. On the *other* hand, the really amazing part is just how much we can do with finite descriptions of functions. This idea that we can write finite descriptions of every Turing machine as a string is going to be integral to the rest of this topic where we explore the limits of what is computable.

Before we get to the limits of computability, that is the limits of what is Turing recognizable, let's first explore the limits of the more restricted notion of decidability.

Our first language is going to be $A_{DFA} = \{(B, w) | B \text{ is a DFA that accepts input string } w\}$. We want to prove that this language is decideable. How do you prove something is decideable? You define a Turing machine for it that never loops. Just as we described how to make a TM that simulates other TMs, we can make a TM that simulates a DFA on an input. Now, since the DFA always accepts or rejects in finite time then the simulation will always accept or reject in finite time. Thus, our decider is simply running the simulation on the input of the text description of B and the input w.

Now, for NFAs we can do a similar thing with $A_{NFA} = \{(B, w) | B \text{ is an NFA that accepts input strin}$ and we can reuse our machine for A_{DFA} by making a new machine that first converts the input NFA to a DFA and then runs A_{DFA} on the resulting DFA and the input string. Since converting the NFA to a DFA is a simple algorithm, we can encode the conversion in a Turing machine, and thus we have our decider.

Also similarly, we can do the same thing for regular expressions and reuse our machine that decides A_{NFA} by first converting the regular expression into an NFA. This way, $A_{REX} = \{(R, w) | R \text{ is a regular expression that generates } w\}$ is also decideable.

We can keep following Sipser pretty closely and talk about other languages related to regular languages that are decideable, but for the moment let's stop here and we'll do more on recognizable languages next time.

13 More Decidability and Recognizability

Continuing from last time, we want to talk more about what kinds of things are decidable and recognizable. We argued semi-formally last time that

- $A_{DFA} = \{(B, w) | B \text{ is a DFA that accepts input string } w\}$

- $A_{REX} = \{(R, w) | R$ is a regular expression that generates $w\}$
- $A_{NFA} = \{(B, w) | B$ is an NFA that accepts input string $w\}$

are all decidable. These tell us if a given DFA/NFA/RegExp accepts a given string, which is basically answering the question "is this string in the this language", so it's pretty easy to see that TMs encompass everything that regular languages could do. Indeed, it's even stronger than that because not only do we have a Turing machine corresponding to each DFA in simulation, but rather we have just *one* Turing machine that is capable of answering that question for *all* regular languages. That's kinda interesting to me, really, that we are now seeing the advantage of describing computation by languages: we can have languages whose elements are descriptions of other computational systems. Now, what about more "meta" properties about regular languages? For example, what if we want to look at a DFA and decide whether or not it was the empty language? It turns out, we can do that with Turing machines. We can't just naively attempt to test all strings and see if none of them are accepted by the DFA: that will lead to a machine that isn't even a Turing recognizer because it will *always* loop. Instead, we'll be a bit more clever (and almost quoting Sipser) by having a machine with the following description:

On input A where A is a DFA

1. Mark the start state of A
2. Mark any state that has a transition coming into it from a state that is marked
3. As long as *a* state was marked in stage 2, go back to stage 2, otherwise go to stage 4
4. If an accept state has been marked, accept, otherwise reject

So in words, what we're doing here is checking to see that there is *some* path from the start to an accept state. We don't care about what this path is, so we're not considering the labels at all but rather just traversing the graph. We'll call the language decided by this E_{DFA}, which is the language of descriptions of DFAs whose corresponding language is empty.

Using this machine that decides E_{DFA} we can make a machine that decides the language $EQ_{DFA} = \{(A, B) | A, B$ are DFAs and $L(A) = L(B)\}$, i.e. a machine that when given two descriptions of DFAs A and B will tell us whether or not the two DFAs decide the same language. The way we do this is with two facts

- Regular languages are closed under complement and intersection
- The symmetric difference of two sets $(M \cap \overline{N}) \cup (\overline{M} \cap N)$ is empty iff the sets are equal. Now why is that? Well the first clause $(M \cap \overline{N})$ is only empty if every string in M is in N, i.e. if M is a subset of N, similarly $(\overline{M} \cap N)$ is only empty if N is a subset of M and thus the union is empty iff both sets are subsets of each other, which is only true if they are equal.

So now what we can do is make a Turing machine that will take two descriptions of DFAs, construct a DFA corresponding to the symmetric difference of the two DFAS, then test to see if it's empty. There's no possibility of loops anywhere in the process, so this machine will be a decider!

What about the context free languages? Well, the equivalent of A_{DFA} above, which is $A_{CFG} = \{(G, w) | G$ is a CFG that generates string $w\}$ is decidable, as is E_{CFG}, but sadly because as has been discussed before context free languages aren't closed under intersection and complement and thus we can't play the same trick as above to get EQ_{CFG} to be decidable. Indeed, it turns out that it *isn't* decidable at all but we have to delay that.

Alright, now we're pretty much done following Sipser so directly. Let's go back to talking about *numbers* for a second. I quoted, without proving it, that the size of the real numbers is much larger than the size of the natural numbers. Let's prove that explicitly in a proof by contradiction, using what's called a diagonalization argument.

If the size, or cardinality, of the real numbers is the same as the natural numbers then we should be able to make a 1-1 correspondence between them, thus labeling the real numbers as the 0th real number, 1st real number, 2nd real number, etc. We can also label each digit of the real numbers starting with the most significant digit as the 0th digit and increasing as we move to the right. We can take these two sets of labels to make a "table", with the rows being all the real numbers "in order" of their label and the columns being the digits. So, for example, at place 3,5 in the table we will find the 6th digit of the 4th real number (we're starting at 0). Now if we have this table, let's make a *new* real number that can't possibily be in the table: it's 0th digit is *something* other than the 0th digit of the 0th number, it's 1st digit is *something* other than the 1st digit of the 1st number, etc. We just arbitrarily pick a number at each digit that meets the criterion.

Now we ask the question? Is this number in the table? If it's not in the table, then it contradicts the assumption that the table contains all real numbers, but if it *is* in the table then it must be some row i, but by the way we constructed this number then it disagrees with the i th real number at the i th digit, which means it would not be equal *to itself.* Hence, there can be no table by this argument.

If this argument makes you slightly uncomfortable, it's okay, don't worry. This argument made Georg Cantor a heretic and an outcast, only for the rest of mathematics to eventually realize he was right in a collective "oops, my bad" after he'd already left the field. It's counterintuitive because we're quantifying over the set of all real numbers when trying to define a real number. If y'all will induldge me briefly, this notion of wreckless quantification in classical set theory is exactly what lead to the development of mathematical logic to prevent such bizarre paradoxes. For example, let's say we have a set of all sets. This is completely allowed under set theory, as strange as it may seem. Does this set contain itself? Well, yes, it would because it contains all sets. Now, however, let's consider the set of all sets that do *not* contain themselves. This is also a perfectly reasonable definition in set theory. Does *it* contain itself? If it does contain itself, then it can't because it only contains sets that don't contain themselves. If it doesn't contain itself, then it does because it contains all sets that do not contain themselves. Either way you try to argue, you get bitten by a nasty contradiction. This is the paradox found by Bertrand Russell that helped motivate the development of ramified type theory, which is itself in a sense the ancestor of types in both modern programming languages and logics.

So why are we even talking about this, other than the fact that it gave me an opportunity to ramble about set theory and the need for mathematical logic? Well we're going to play a very similar trick to this table in order to show that the language $A_{TM} = \{(M, w) | M \text{ accepts the string } w\}$ isn't decideable. It's still recognizable though, which we can see if we consider our Turing machine simulator from last time: that machine would accept if M accepted w, reject if it rejects, and loop if M loops on w.

Let's start with the proof by contradiction: assume there is a Turing machine A that decides A_{TM}, then we can make a Turing machine D out of A as follows

On input M, where M is a description of a Turing machine

1. Run A on the input (M, M), i.e. if M accepts on its own description

2. If A accepts, reject, and if A rejects, then accept

Okay, neato, right? Well we know that the set of all Turing machines is countable, so let's make a table labeled by all the Turing machines along both axes. Each entry in the table, (i,j), will be the output of the i th Turing machine when run on the description of the j th Turing machine, i.e. the output of A on the pair of M_i and M_j. Well D must appear somewhere on the table and have some index, let's call it n, so what do we find at entry (n, n)? If it's an accept, then that means D accepts when passed D, except that D *rejects* on D when D applied to D accepts. Similarly, if D applied to D rejects then that means it must accept. Either way we got a contradiction, which ultimately means the table can't exist, which means that A doesn't exist and there is *no* decider for A_{TM}.

Next time, we'll cover computable reductions and do a whole lot more with proving languages decidable, undecidable, recognizable, or unrecognizable. It'll be fun!

14 Recognizability and Computable Reductions

We return again with more on Turing recognizability and introducing the tools we need for showing that languages are decidable, recognizable, or neither.

First off, we showed that there are some languages that aren't decidable last time. We've argued before, informally, that there *must* be languages that can't be described by Turing machines at all and thus aren't even recognizable. Here we'll give a nice concrete example that also will demonstrate that Turing machines aren't closed under complement.

Recall the language $A_{TM} = \{(M, w)|M \text{ accepts the string } w\}$, now consider it's complement $\overline{A_{TM}} = \{(M, w)|M \text{ rejects or loops on the string } w\}$. Intuitively, would we expect this to not be recognizable. Why? Because recognizing A_{TM} amounts to just simulating the action of M on w. That's easy enough, because you can just sit and wait for an answer. If you get an accept eventually, then you can accept, but if the simulation loops well it's okay to loop yourself because you can either fail *or* loop on a rejection. If you're trying to simulate the complement, however, you hit an awkward problem: how do you sit around and wait forever on the loop and somehow *still* return an accept? You might object and say that there must be a more clever way to do this simulation and you'd be right, we didn't write a proof we just made an appeal to intuition which is why we'll do a formal proof that $\overline{A_{TM}}$ is unrecognizable after we introduce one more fact.

The lemma in question is the fact that if both a language and it's complement are recognizable, then the language is actually decideable. Look at it this way, we have a language L that is recognizable then that means there will be a machine that returns "accept" in finite time whenever a string w

is in L. The complement being recognizable means that there will be some machine that returns "accept" in finite time whenever a string w is in $\overline{L}$. Now, we can make a new machine that combines these two recognizers in a multitape dual simulation, where it alternates taking a step in one machine with taking a step in the other. This way, no matter whether a string is or isn't in the language, then we'll get an "accept" back from one of the two machines in finite time. As long as we reject when the complement machine accepts, then we have a decider for the language.

This means, however, that if a language isn't decidable but is recognizable, then it's complement cannot be recognizable. Clearly, A_{TM} is such a language and thus $\overline{A_{TM}}$ is not recognizable.

Great, now that we have concrete proven examples of both undecidable and unrecognizable languages now we can introduce the notion of a computable function whose definition we're pretty much just going to steal wholesale from Sipser: a function $f : \Sigma^* \to \Sigma^*$ is computable if there exists some Turing machine M such that on every input w, M halts with just $f(w)$ on the tape. Now, *now* Turing machines are starting to resemble computation as we normally think of it, aren't they? We're introducing this construction, though, to define the notion of a *mapping reduction* from one language to another.

Given languages A and B, A is reducible to B, written as $A \leq_m B$, if there is a computable function f such that if a string $w \in A$ then $f(w) \in B$ and if $w \notin A$ then $f(w) \notin B$, in other words $f(w) \in B$ iff $w \in A$. What does this mean intuitively, though? It means that we can take the question "is w in A " and turn it into the question "is $f(w)$ in B ", because by the condition that $f(w) \in B$ iff $w \in A$ these questions are equivalent to each other. If you answer one, you've answered the other. Which means, in turn, that if we already know how to answer "is s in B " then we automatically have a way to answer "is w in A " without having to even build a machine for it, we only need the reduction.

We've seen this already, actually, because this is in essence what we did with the difference between A_{DFA}, A_{NFA}, and A_{RegExp}. Remember how, as we defined each of these machines in turn we just turned an NFA into a DFA or a RegExp into an NFA and then used the machine we defined previously. This is also how we defined EQ_{DFA}, by *reducing* the problem to E_{DFA}.

There's a couple of really nice properties about reductions. First, that if $A \leq_m B$ and B is decidable then we know that A is decidable. We know this because our decider for A is just to compute the reduction to B and then run the decider for B. This also means, however, that if $A \leq_m B$ and A is *undecidable* then B must be undecidable as well or else we'd get a contradiction.

Similarly for recognizable languages, if $A \leq_m B$ and B is recognizable then so is A and if A is unrecognizable then so to must B.

I think I'll leave things here for now and next post will be lots and lots of examples of reductions because it's a cool technique that's worth savoring.

15 More on Reductions

Let's go back and redo our results about regular languages except in terms of reductions. We'll try to be very formal for this section. To start with, we recall the machine that decides the language $A_{DFA} = \{(M,w)|M \text{ accepts the string } w\}$, which is

On input (M,w) where M is a DFA then

1. Simulate the DFA M on input w
2. If the simulation accepts, accept. If the simulation rejects, reject.

Now, to prove that A_{NFA} is decidable we need to give a reduction of it to A_{DFA}, i.e. prove $A_{NFA} \leq_m A_{DFA}$. To do this we need to make a Turing machine that, when given a pair (N,w) where N is an NFA and w is the input string turns that into a pair (M,w) where M is a DFA and where M accepts w iff N accepts w. We already know this is possible, though, because the power-set construction for turning an NFA into a DFA is definitely a computable function. So our reduction will will look

On input (N,w) where N is an NFA:

1. Convert N to a DFA we'll call M
2. Write the description of M and w onto the tape
3. If N is not an NFA we blank out the tape and halt

So for this function, is $(N,w) \in A_{NFA}$ iff $(M,w) \in A_{NFA}$? Yes! We know that because the powerset construction creates a DFA that decides the exact same language as the original NFA. Just for being truly rigorous we've defined what happens if we *aren't* given an NFA. We just need to do this to make sure that our computable function sends a string into A_{DFA} iff that string is an element of A_{NFA}.

What about a more complicated example? Consider the language $H = \{(M,w)|M \text{ halts on the string } w\}$. This is subtly different than A_{TM}, since it's not asking whether the Turing machine *accepts* the string but rather if the Turing machine either accepts or rejects in finite time. Is this language decidable? We could do a very similar argument to our previous one about A_{TM} but we can do this more easily by using a reduction. Recall last time when we stated that if $A \leq_m B$ and A is undecidable then B must be undecidable. In this case, we already know that A_{TM} is undecidable so if we can reduce A_{TM} to H then we win. A reduction in this case means that we need to take a pair (M, w) and map it to a pair (N, w) such that $(N, w) \in H$ iff $(M, w) \in A_{TM}$. We'll do this as follows

On input (M, w) where M is a Turing machine:

1. Build a description for a new TM R whose definition is On input s then
   (a) Simulate M on s
   (b) If it accepts, accept, if it rejects loop
2. Write the pair (R, w) to the tape

We can see that the Turing machine R will halt on a string iff the Turing machine M will accept the string. This means that $(R, w) \in H$ iff $(M, w) \in A_{TM}$ which is exactly what we needed. Thus, H is undecidable.

What are some other languages we can prove undecidable this way? Well, let's prove that $E_{TM} = \{M | L(M) = \emptyset\}$ is undecidable and, indeed, unrecognizable by showing that we can reduce $\overline{A_{TM}}$ to it. This means, again, that we need to turn a pair (M, w) where M doesn't accept w into a TM N where $L(N)$ is empty. The way we'll do this is much like our previous example, where we build a *new* Turing machine that has the property we want and pass that on as input. So our computable reduction will be

On input (M, w) where M is a Turing machine:

1. Build a description for a new TM R whose definition is On input s then
   (a) if M accepts w then accept else reject
2. Write the description of R to the tape

We can see that this is a Turing machine that, if M accepts w then the resulting R will not be the empty language, but if M rejects w or loops then the language will be empty and thus an element of E_{TM}. This means that we have a proper reduction of $\overline{A_{TM}}$ to E_{TM} and thus E_{TM} cannot be recognizable and thus not decidable either.

Now, when I first thought about this problem I wanted to do a reduction of A_{TM} to E_{TM} as follows:

On input (M, w) where M is a Turing machine:

1. Build a description for a new TM R whose definition is On input s then
   (a) if M accepts w then reject else accept
2. Write the description of R to the tape

and that certainly seemed reasonable at first, because if (M, w) is in A_{TM} then the language for R will be empty. Ah, but this is where the iff part of things is important because while things are all well and good if M actually *rejects* the string w, iff M *loops* on w then $L(R)$ is going to be the empty language even though M didn't accept the string. Oops!

16 Time Complexity

So in this post we're going to try and give a smattering of the important parts about time complexity that will actually be covered in lecture. This will be fairly high-level and overview-y because, honestly, most of the time this course never gets to this material anyway so I figure even just spending an entire two-hour lecture on it is doing better than typical. This also leaves room for a solid two-hour introduction to the lambda calculus as a final treat for those interested and a lecture off for those who ain't.

All that being said, let's begin.

So until this point we've been splitting languages into three categories—decidable, recognizable, and unrecognizable—based upon the existence of a Turing machine that can decide or recognize the language. We've also shown that, with respect to this stratification of languages, there is no real difference between deterministic and non-deterministic Turing machines. This categorification is useful from the perspective of figuring out *what a computer can do*, but there's another important set of questions in Computer Science: what can a computer do *feasibly*?

Of course, it depends on what we mean by "feasible". Does "feasible" mean "finishes faster than it takes to drink a cup of coffee"? Does it mean "takes less than a gig of RAM"? When we talk about feasible and unfeasible problems in Computer Science what we mean is "scalable", because in general even the most complex of problems are able to be handled in a short time if the specific instance of the problem is "small". Of course, what do we mean by small? When we're talking about things like sorting of lists, it's a bit obvious that a small instance of the problem is going to be one with a short list. There are other problems where the size of the input may not be as intuitively obvious. Since we're dealing with Turing machines though there's a fairly obvious way we can define size. The size of the input to a Turing machine is simply the length of the input string on the tape. Also, for the remainder of this discussion we're only going to talk about decidable languages. If a language is not decidable, then we're allowing for the possibility that it could run forever on an input with us, the user, having no way of actually knowing if it's still-working-but-taking-a-long-time or if it's stuck. We rule out that as a possibility for convenient computational use and only stick to things that are decidable and compute in a finite time.

Now there are two different measures we could be using for scalability: time and space. We're really only going to discuss time, not because space usage isn't also important but in a sense time usage is the more important one. We're getting better every year at having more memory with which to work on our computers, but we haven't yet figured out how to cram more seconds into a day. We can use swap space and give ourselves massive amounts of memory (to a point, before we just slow ourselves down to a crawl) but better and better processors don't buy you much when your algorithms scale poorly. Of course, what do we mean by scale poorly? To answer that question, we'll introduce the notion of time complexity and "big O" notation that you're probably familiar with but we should review anyway.

The "time" that a Turing machine takes on an input is the number of steps it runs on that input before it terminates. Again, we're dealing only with decidable languages so that it will **always** terminate in finite time. We're just trying to describe how finite is finite. We say that a Turing machine M 's running time is described by a function $f : \mathbb{N} \to \mathbb{N}$ if the maximum number of steps that M takes on an input of length n is $f(n)$. In other words, for every length n, $f(n)$ gives us the maximum number of steps that the machine takes for an input of that length. Now, ultimately, if we're talking about scalability we don't really care about constant factors so we want to treate $f(n) = 2 * n$ and $f(n) = 3 * n$ the same, as well as $f(n) = n + 10$. This might seem silly because maybe you would care about a constant difference of 10 or 100, maybe that feels like it should change a problem from feasible to unfeasible but look at it this way: we're trying to distinguish problems that are in principle computable in a reasonable time as the input grows, and a factor of 10 or even 100 could be overcome at some point by improved hardware, from problems whose number of steps grows exponentially from small to longer-than-the-lifetime-of-the-universe just by making the input 10 times longer.

To the end of making this distinction, we define that for $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{R}^+$ that $f(n) = O(g(n))$ when there exists positive integers c and n_0 such that $f(n) \leq c * g(n)$ for all $n > n_0$. In other words, $f(n)$ is, for sufficiently large n always bounded by $g(n)$ up to some constant factor.

This means that we can finally describe the time complexity class of a function t, TIME($t(n)$), to be all languages that can be decided by a Turing machine that runs in $O(t(n))$ time. Note that this is cumulative in the sense that if a TM runs in $O(t(n))$ time and $t(n) < t'(n)$ for all n, then we have that the TM runs in $O(t'(n))$ time as well. This little detail might be important to those of you paying close attention to how we define the complexity classes P and NP shortly.

The problems we'll call "feasible" in time complexity are the ones that run in polynomial time, i.e. TIME(n^k) for some k, since those are the ones we can reasonably expect to scale well, more formally we have that $P = \bigcup_k \text{TIME}(n^k)$. Now, there a two ways to prove that a language A belongs to P. We can either find a Turing machine that decides A and then check it's complexity to show that it's $O(n^k)$ for some k *or we can give a reduction to a language we already know is polynomial*. Why? Because if we have a language A, a reduction $A \leq_m B$, and know that $B \in P$ then we can build a decider for A automatically that will have time complexity $O(f_R + n^b)$ where f_R is the time complexity of the reduction and b is the polynomial factor for the decider of B. Ah, but wait, there's a slight problem here! What about the complexity f_R? If that's not polynomial as well then we've blown the whole thing out of the water. To this end, we insist that the reduction $A \leq_m B$ run in polynomial time as well and we'll use the notation $A \leq_p B$ to mean that there exists a *polynomial time* reduction of A to B.

This is all well and good, but now we come back to the question of the difference between deterministic and non-deterministic Turing machines. To this end, we define the complexity class NP which are the problems that can be solved in polynomial time *by a non-deterministic TM*. How do we define the time taken by a non-deterministic machine, though? Well, since we're dealing only with *deciders* which, for non-deterministic machines, means that every single possible computational path terminates in finite time then we can just choose the "number of steps" taken by the machine to be the longest number of steps of any of the branches. This encapsulates the intuition we've been building for what a non-deterministic machine does in the first place: it tries all possible computational paths simultaneously and when they're all done computing, the machine is done with the input. Now, every non-deterministic machine can be turned into a deterministic machine, which we know because we've seen the construction before. What does this do to the running time? Well, if a non-deterministic machine runs in time $O(t(n))$ then the naive conversion to a deterministic machine that simulates the non-determinism is going to run in $O(2^{t(n)})$ time, which is exponentially longer.

Does this mean, then, that polynomial-time non-deterministic machines necessarily have no polynomial-time determiminstic counterparts? Certainly not! A long-standing problem in computer science is if $P = NP$, or in other words, if for every language that can be decided with a polynomial *non-deterministic* machine then there exists a polynomial *deterministic* machine that decides the same language. This is an open question, but informally I think most people would be surprised if the answer was yes. Non-determinism buys you so much when it comes to Turing machines that, honestly, it would be counterintuitive if it was never necessary for efficiency.

Now, it might be obvious the kinds of things that are in P: sorting algorithms, binomial search, etc. that you've dealt with in intro programming courses but what kinds of problems are in NP? We'll consider a particular, useful example: the clique problem.

Formally the language CLIQUE is defined as CLIQUE $= \{(G, k) | G$ is a graph with a clique of size k but what exactly is a clique? A clique is a subgraph where every node in the subgraph has an edge to every other node in the subgraph. Now, there is no known way to, in polynomial time with a deterministic machine, to determine if a given graph has a clique of a given size. On the other hand, it's very easy to show that CLIQUE $\in NP$. Our machine simply simultaneously checks all possible subgraphs of size k or greater, which means that the longest possible thing to check would be the entire graph itself and thus the algorithm is n^2 for a non-deterministic machine. Fantastic!

There's another way we can think of NP, though, that's a bit more useful in terms of proving that languages are in NP by construction. We equivalently say that a language is in NP when there exists a polynomial time *verifier* for the language, which is a machine that takes

- a proposed element of the language
- a piece of data, called a certificate, meant to witness the verification

and is allowed to use both of them in determining if the object is in the language. For example, a polynomial time verified for CLIQUE is a machine that takes a pair of a graph and a set of nodes in the graph that should correspond to the required clique. The verifier simply checks that the subgraph is a clique, which is polynomial in the size of the input.

How is this equivalent to our previous definition? Well, we need to then construct a polynomial non-deterministic Turing machine from a polynomial verifier and then visa-versa. If we have a verifier, we can construct a non-deterministic machine by having the non-deterministic machine simultaneously guess at all possible certificates and then run the verifier. If we have a non-deterministic machine, then we can make a verifier by using the correct path through the computational tree of the machine as the certificate. (I'm eliding some details, obviously)

The final topic I want to hit briefly is the idea of NP-hard and NP-complete problems. Essentially, a NP-hard problem is one that *every* NP problem can be reduced to in polynomial time, an NP-complete problem is an NP-hard problem that is also in NP. This seems like an unusual property to have, but it turns out that there really are NP-hard and NP-complete problems. CLIQUE, as discussed above, is actually an NP-complete problem. This means two important things: first, that if there is a way to solve CLIQUE with a polynomial deterministic machine then $P = NP$ and second, that an easy way to check if a problem is NP is to try reducing it to CLIQUE.

I think that's pretty much all I'm hoping to cover in this very very sketchy overview of time complexity. The one lecture after this will be the untyped lambda calculus, just for fun.

17 The Lambda calculus

Having seen Turing machines and talked a bit about what they are, what it means for a language to be Turing recognizable or decideable, and even spending a little bit of time talking about time complexity for Turing machines, let's finish out the course with a bit on a different notion of computation that is equivalent to the Turing machines: the lambda calculus.

The lambda calculus was, essentially, invented by Alonzo Church for investigations into constructive logic but while it turned out that the untyped lambda calculus, as a proof theory for a logic, is entirely inconsistent and unsuited to the task it *did* turn out that the lambda calculus describes all computable functions in the same way that the Turing machines do. We know this for certain because we can encode any Turing machine as a program in the lambda calculus and visa-versa, so they have equivalent power. The lambda calculus, though, has a very different character than Turing machines. While Turing machines were obviously finitary machines that had a nice, physical, interpretation that intuitively makes sense as a computable process, the lambda calculus is more akin to a real programming language whose rules, while simple, aren't obviously realizable in a computable way. (When I say "obviously realizable" I mean that it's not clear just by glancing at the rules that there isn't anything non-computable going on, since we know quite well how to interpret the rules computably as there are many many ways to write a program that is an interpreter for the lambda calculus. I've written at least a dozen such interpreters so far.)

Conceptually, in the lambda calculus everything is a function. Well, almost. Everything is a function or a variable. There are only two things you can actually *do* in the lambda calculus: you can make a new term out of another term with *lambda abstraction*, which is how you make functions, or you can apply a term to another term. If you're used to most programming languages, the "apply any term to any other term" might seem kinda weird, but in this case it's okay because Everything is a Function and thus every term is capable of accepting a term as an argument.

We'll describe the lambda calculus as an inductive structure, much like we did with RegExps so long ago. First, though, we fix a countably infinite set that we'll call V, the variables of the calculus. The inductive type of syntax for the lambda calculus, l, is thus built up by the following rules

- $\lambda x.l$, for $x \in V$
- $l_1 l_2$
- y where $y \in V$

and that's it, we can all go home. Oh, sorry, right we need to make sure these rules actually mean something. First, let's say in words what these syntactic forms are actually going to mean. The very first one says that if we have a term in the lambda calculus, l, which we can think of as the body of the function then we can wrap it up in the *lambda abstraction* and bind the variable x to be the argument of the function. This means that whenever we pass an argument to the function $\lambda x.l$ then the argument will be *substituted* in for x everywhere in l. We'll make more rigorous the notion of substitution soon. The second syntactic form means that a lambda calculus term can be made up of applying one term to another. The final syntactic form means that a variable is a term in the lambda calculus.

Perhaps some examples are in order. Recall how we define functions in, say, high school mathematics where we have things like $f(x) = 3 * x$. Well, what exactly *is* the act of saying $f(x) = \ldots$? We've been using this notation so much over our lives I'm not sure how often it occurs to us that there *should be* a formal meaning to the syntax. The lambda calculus gives us a way of describing what $f(x) = \ldots$ means. It's really a shorthand for saying that $f = \lambda x. \ldots$, i.e. it's giving a name to an act of lambda abstraction. Now, the bare untyped lambda calculus doesn't *have* a way to give names to terms like that but we can fake any use of names just by using *more lambda abstractions*. In this case, even just saying $f = \lambda x. \ldots$ could be considered short hand for \$(λ f. rest of program) (λ x. . . .).

The act of substituting in a value into a function, like when we say that $f(3) = 3*3$ given our f above, is how we actually *evaluate* the act of applying one lambda calculus term to another lambda calculus term. Again, this is something we've been doing so long in math classes we might have learned to not question what exactly it means to substitute the arguments into the body of a function. In the lambda calculus, we'll represent substitution syntactically as $l[v/x]$, which in words means that we substitute v in for x in the term l.

Defining substitution over the syntax of lambda calculus terms might seem pretty simple. Let's give it a shot!

- $y[v/x] \rightarrow y$ where $y \neq x$
- $x[v/x] \rightarrow v$
- $(l_1 l_2)[v/x] \rightarrow (l_1[v/x])(l_2[v/x])$
- $(\lambda y.l)[v/x] \rightarrow \lambda y.l[v/x]$

Okay, well this seems really neat and simple. Now we can also inductively define how evaluation of lambda terms works which goes a lil' something like this:

- $(l_1 l_2) \rightarrow (l'_1 l_2)$ if $l_1 \rightarrow l'_1$
- $(l_1 l_2) \rightarrow (l_1 l'_2)$ if $l_2 \rightarrow l'_2$
- $(\lambda x.l) \rightarrow (\lambda x.l')$ if $l \rightarrow l'$
- $(\lambda x.l_1)l_2 \rightarrow l_1[l_2/x]$

That's really all we need to evaluate the lambda calculus. Since this all looks pretty simple let's just try a few examples.

- $(\lambda x.x)(\lambda y.y) \rightarrow x[(\lambda y.y)/x \rightarrow \lambda y.y$

- $(\lambda z.zz)x \rightarrow xx$

Okay, looking good so far, so let's try one more example that should *totally work* because I haven't been setting this up at all for a weird surprise:

- $(\lambda x.\lambda y.xy)y \rightarrow (\lambda y.xy)[y/x] \rightarrow \lambda y.yy$

But this is clearly *wrong* because if we just used a different variable for the binding in the lambda abstraction then it would have been

- $(\lambda x.\lambda z.xz)y \rightarrow (\lambda z.xz)[y/x] \rightarrow \lambda z.yz$

which doesn't behave even remotely the same way. So we're left in the rubble of a fallen calculus, clutching it to our collective chest and crying out "whhhhyyyyyyyyy".

Wait, no, sorry there's a way to fix this. We make use of the fact that $(\lambda x.l) = (\lambda y.l[y/x])$ whenever y isn't already bound in l, basically because the particular variable chosen for a binding site doesn't actually matter. In other words, $f(x) = 3 * x$ is the exact same function as $f(y) = 3 * y$.

To this end, we redefine our notion of substitution to be

- $(\lambda y.l)[v/x] \rightarrow \lambda y.l[v/x]$ where y must be different than x, and if $y = x$ then we perform a variable renaming of y to some other variable that is different any variable bound in l or x. This is what we technically call capture-avoiding substitution.

With substitution fixed, we now have our evaluation of the untyped lambda calculus defined. What's really the point, though? We don't have any data types, just a way to build functions that can return other functions. This is where the magic happens: it turns out that with only functions we can actually fake any other kind of datatype that we'd want.

Let's take the simplest possible example: booleans. We're going to play a little bit of a trick and think of booleans by what they *do* not what they *contain.* At it's heart, what is a boolean value but a way to do branching? If you have truth, then you take one branch and if you have false you take the other branch. How would we encode this in the lambda calculus? All we have is functions, so let's have functions that take two arguments, branches, and return one or the other

- $t = \lambda xy.x$

- $f = \lambda xy.y$

which means that really an if-statement is just running the boolean

- if$_{\text{thenelse}\_}$ \$ $=$ λ b x y. b x y\$

Now, in order to test that these really do represent booleans we need to be able to define boolean operations on them such as $\wedge$ and $\vee$ and not. Defining not is easiest so let’s start with that. Not should take a boolean and return the opposite value, or in terms of booleans-as-branching it should flip which branch is taken.

- not $= \lambda bxy.byx$

similarly, since we know that "and" should branch ’true’ if both arguments are true and false otherwise, which if we think of how we’d implement as branching with nested if-statements will look something like

- $\wedge = \lambda b_1 b_2 xy.b_1(b_2 xy)y$

and or follows very similar logic

- $\vee = \lambda b_1 b_2 xy.b_1 x(b_2 xy)$

So we’ve demonstrated that we have a notion of booleans with only functions. What else can we build? Well, let’s try natural numbers. First, though, we need to think about what natural numbers are in terms of behavior. That might seem a bit odd: numbers are just things we add and subtract etc., right? Well, no, at least not natural numbers. Natural numbers are a way to count things, a way to repeat behaviors a certain number of times, in other words natural numbers are a kind of for-loop. A natural number in mathematics is either zero or one plus another natural number. A natural number "behaviorally" is either just returning the base case of the recursion, or performing an operation on the recursive argument. To this end we define

- $0 = \lambda sz.z$
- $1 = \lambda sz.sz$
- $2 = \lambda sz.s(sz)$

etc. or in other words we just have

- $0 = \lambda sz.z <$- base case

- $suc = \lambda nsz.s(nsz)$ <- recursive case

but let's make a quick check to ensure we've done this right

- $suc(0) = (\lambda nsz.s(nsz))0 \rightarrow \lambda sz.s(0sz) \rightarrow \lambda sz.sz$
- $suc(suc(0)) = (\lambda nsz.s(nsz))(suc(0)) \rightarrow (\lambda sz.s(suc(0)sz)) \rightarrow (\lambda sz.s(sz)) = 2$

We can even define operations such as +,*, or - on these numbers that will do the right thing. For example, + can be defined as

- $+ = \lambda n_1 n_2 sz.n_1 suc(n_2 sz)$ or in other words that adding n_1 and n_2 is given by apply adding 1 to n_2 a total of n_1 times.

The one last topic before we wrap up this somewhat informal treatment of the untyped lambda calculus is how we do recursion in general. We know that we need the ability to define arbitrary recursion because this calculus is equivalent to Turing machines. We do this using what's called the Y combinator, which is defined as

- $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$

This might be a very intimidating and confusing term when we first look at it, but in the end it satisfies a very important property, namely that $Y(f) = f(Y(f))$ which we can prove directly

- $Y(f) = (\lambda x.f(xx))(\lambda x.f(xx)) \rightarrow f((\lambda x.f(xx))(\lambda x.f(xx))) = f(Y(f))$

In other words, the trick of apply terms to themselves gives us this weird self-replicating feature of the Y-combinator. Of course, the Y combinator will just be an infinite loop unless the function f has some kind of base case that short-circuits out.

We'll try a basic example of using the Y-combinator after we introduce a few more pieces

- $* = \lambda n_1 n_2 sz.n_1(n_2 s)z$
- \$isZero = λ n. n (λ x. f) t \$ <- the trick here is that if n is 0 then it'll just return it's second argument which will be t

Now we can define the factorial function with the Y combinator

- $fact'(f, n) = \lambda f.\lambda n.isZero(n)1(n * (f(n-1)))$
- $fact(n) = Y(fact')(n)$

So I think we'll go ahead and leave this here because I'm not sure if we'll even manage to cover all of this is a single lecture!